Efficient and expressive bytecode-level instrumentation for Java programs

Chukri Soueidi1 · Marius Monnier1 · Yliès Falcone1

Accepted: 27 October 2021 / Published online: 29 June 2023 © The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2023

Abstract
We present an efficient and expressive tool for the instrumentation of Java programs at the bytecode level. BISM (Bytecode-Level Instrumentation for Software Monitoring) is a lightweight Java bytecode instrumentation tool that features an expressive high-level control-flow-aware instrumentation language. The instrumentation language is inspired by the aspect-oriented programming paradigm in modularizing instrumentation into separate transformers that encapsulate joinpoint selection and advice inlining, that is, the selection of points of interest in the execution and the execution of additional code at these points, respectively. BISM allows capturing joinpoints ranging from bytecode instructions to methods execution and provides comprehensive static and dynamic context information. It runs in two instrumentation modes: build-time and load-time. BISM also provides a mechanism to compose transformers and automatically detect when they interfere on the base program. Transformers in a composition can control the visibility of their advice and other instructions from the base program. We show several example applications for BISM and demonstrate its effectiveness using three experiments: a security scenario, a financial transaction system, and a general runtime verification case. The results show that (i) BISM instrumentation incurs low runtime and memory overheads, and (ii) the code produced by BISM performs better than the one produced by existing Java instrumentation tools.

Keywords Instrumentation · Runtime verification · Monitoring · Java · Bytecode · Aspect-oriented programming · Control flow · Static and dynamic contexts

1 Introduction

Instrumentation is essential in software engineering and verification activities. In the software monitoring and runtime verification workflow [2, 14, 17], instrumentation allows extracting information from a running software to abstract the execution into a trace fed to a monitor. Depending on the information needed by the monitor, the granularity level of captured and extracted information may range from coarse (e.g., a function call) to fine (e.g., an assignment to a local function/method variable, a jump in the control flow).

General-purpose tools for instrumenting Java programs have two important aspects that distinguish their usability: their level of expressiveness and level of abstraction. Expressiveness determines how much the user can extract information from the bytecode and alter the program’s execution. Moreover, abstraction determines how much the user has to deal with bytecode details and how high-level the code has to be written. Extracting fine-grained events requires tools with high expressiveness. However, these tools are often too low-level and require expertise on the bytecode, making the instrumentation too verbose and error-prone. An important requirement of an instrumentation process or tool is to have a low-performance impact on the target program, which can be measured by the runtime overhead of the inserted code, the memory overhead, and the size of the generated executable.

Aspect-oriented programming (AOP) [24] is a popular paradigm for instrumenting a program. AOP advocates the modularization of crosscutting concerns such as instrumentation. For Java programs, runtime verification tools [3, 15] have long relied on AspectJ [25], which is one of the most adopted AOP implementations for Java. At the core of AOP is the joinpoint model, which consists of joinpoints (points in
the execution of a program) and pointcuts (a mechanism to select joinpoints). AspectJ provides a high-level joinpoint model for convenient instrumentation. Although AspectJ provides several functionalities oriented on the development process and program structure, these are seldom used in the context of verification. Moreover, AspectJ does not offer enough flexibility to capture and extract fine-grained information from the program. This makes instrumentation tasks that require capturing low-level bytecode regions, such as bytecode instructions, local variables of a method, and basic blocks in the control-flow graph (CFG), unachievable with AspectJ. In Sect. 8.1, we demonstrate an instrumentation scenario that cannot be achieved with AspectJ. Furthermore, AspectJ provides limited support for writing program static analyzers that can be combined with runtime verification. In particular, the provided static crosscutting constructs are only limited to inter-type declarations, weave-time error, and warning declarations, and exception softening. These constructs are not expressive enough to allow users defining complex compile-time computations.

Nevertheless, there are several low-level bytecode manipulation frameworks such as ASM [7] (in reference to the __asm__ C++ operator) and BCEL [1] (Byte Code Engineering Library) which are highly efficient and expressive. However, writing instrumentation in such frameworks is too verbose, tedious, and requires expertise in bytecode. DiSL [27] is a bytecode instrumentation framework that addresses these problems in that it enables flexible low-level instrumentation and, at the same time, provides a high-level language. However, DiSL does not support inserting bytecode instructions directly, but allows writing custom transformers that allow the user to traverse the bytecode freely and modify it. These custom transformers can be run only after the main DiSL instrumentation process, and the developer needs to revert to low-level bytecode manipulation frameworks to implement them. This makes several scenarios tedious to implement in DiSL and requires workarounds that incur a considerable bytecode overhead. For example, in Sect. 8.1, we demonstrate the overhead incurred by DiSL when instrumenting an inline monitor that duplicates if-statements in a program.

Contributions We introduce BISM (Bytecode-Level Instrumentation for Software Monitoring), a lightweight bytecode instrumentation tool for Java programs that features an expressive high-level instrumentation language. BISM is designed to accommodate users who want to instrument programs without dealing with bytecode details, as well as advanced users who are proficient in the bytecode. Its language is inspired by AOP, but adopts an instrumentation model directed towards software monitoring and runtime verification. In particular, BISM provides separate classes, transformers, that encapsulate joinpoint selection and advice inlining. It offers various instrumentation selectors that select joinpoints covering bytecode instructions, basic blocks, and methods execution. It also provides access to a set of comprehensive joinpoint-related static and dynamic contexts to retrieve relevant information. BISM provides a set of advice methods that specify the advice to insert code, invoke methods, and print information.

BISM is control-flow aware. That is, it generates CFGs for all methods and provides access to them in its language. Moreover, it 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, for instance, in the security domain, to check for control-flow integrity. BISM also provides a mechanism to compose multiple transformers and automatically detect their collision in the base program. Transformers in a composition are capable of controlling the visibility of their advice and other original instructions. BISM is a standalone tool implemented in Java that requires no installation, as compared to AspectJ and DiSL. BISM can run in two instrumentation modes: build-time, to statically instrument a Java class or Jar file, and load-time, to run as a Java agent that instruments classes being loaded by a running JVM.

We show several applications for BISM in static and dynamic analysis of programs. We also demonstrate BISM effectiveness and performance using three complementary experiments. The first experiment shows how to instrument a program to detect test inversion attacks. For this purpose, we instrument an AES (Advanced Encryption Standard) implementation. The second and third experiments show how to instrument in general runtime verification cases. In the second experiment, we instrument a simplified financial transaction system to check for various properties from [3]. In the third experiment, we instrument seven applications from the DaCapo benchmark [6] to verify the classical HasNext, UnsafeIterator and SafeSyncMap properties. We compare the performance of BISM, DiSL, and AspectJ in build-time and load-time instrumentation, using three metrics: size, memory footprint, and execution time. In build-time instrumentation, the results show that the instrumented code produced by BISM is smaller, incurs less overhead, and its execution incurs less memory footprint. In load-time instrumentation, the load-time weaving and the execution of the instrumented code are faster with BISM.

A much shorter version of this paper appeared in the proceedings of the 20th Runtime Verification conference [31]. The paper provides the following additional contributions:

- new language features: meta selectors, support for synthetic local arrays, and configuration files (in Sect. 3);
- instrumentation model, underlying shadows and equivalence (in Sect. 4 and Sect. 4.4);
- transformer composition, collision detection and visibility control of advice (in Sect. 5);
Fig. 1  BISM overview

- use cases to demonstrate the usage of BISM in different contexts (in Sect. 6);
- an additional experiment demonstrating BISM effectiveness using a benchmark from the competitions on runtime verification \[3, 30\] (in Sect. 8.2).

Paper organization The rest of this paper is organized as follows. Section 2.2 overviews the design goals and the features of BISM. Section 3 introduces the language of BISM. Section 4, details the instrumentation model of BISM. Section 5 shows transformers composition in BISM. Section 6 presents some use cases of BISM in various contexts. Section 7 presents the implementation of BISM. Section 8 reports on the case studies and a comparison between BISM, DiSL, and AspectJ. Section 9 discusses related work. Section 10 draws conclusions.

2 BISM in a nutshell

In this section, we overview BISM and discuss the design goals and its objectives.

2.1 Overview

BISM is a bytecode instrumentation tool for Java programs implemented on top of ASM [7]. Figure 1 shows a high-level overview of BISM. The user provides: the base program, and the instrumentation logic written in transformers with BISM language (see Sect. 3). BISM encapsulates and performs 3 main steps that, in general, can describe any instrumentation task. It parses the bytecode to obtain a representation of the base program, then generates the needed transformations that are specified by the user in transformers, and finally weaves those transformations into the base program to obtain an instrumented program. In Sect. 7, we provide more details of the BISM instrumentation workflow.

2.2 Design goals and features

BISM is a tool on which RV tools can rely to perform efficient and expressive instrumentation. In this section, we describe the design goals and features of BISM.

Simple instrumentation model BISM provides an instrumentation model that is easy to understand and use (see Sect. 4). In particular, it does not have the full notion of pointcuts that allows specifying a predicate to match different joinpoints. Instead, it offers a fixed set of selectors that capture granular joinpoints. Each selector is associated with a well-defined region in the bytecode, such as a single instruction, basic block, control-flow branch, or method. We believe that the selectors in BISM can capture the most used joinpoints in the runtime verification of programs executing on the JVM.

Instrumentation mechanism BISM provides a mechanism to write separate instrumentation classes in standard Java. An instrumentation class in BISM, which we refer to as a transformer, encapsulates the instrumentation logic, that is, the joinpoint selection and the advice to be injected into the base program. Advice is specified using special advice instrumentation methods provided by the BISM language that allow bytecode insertion, method invocation, and printing.

Access to program context BISM offers access to complete static information about instructions, basic blocks, methods, and classes. It also offers dynamic context objects that provide access to values that are only available at runtime, such as local variables, stack values, method arguments, and results. Moreover, BISM allows accessing instances and static fields of these objects. Furthermore, new local variables and arrays can be created within the scope of a method to pass values needed for instrumentation.

Control-flow context BISM generates the CFGs of target methods out-of-the-box and offers this information to the user. In addition to basic block entry and exit selectors, BISM provides specific control-flow related selectors to capture conditional jump branches. Moreover, it provides a variety of control-flow properties within the static context objects. For example, it is possible to traverse the CFG of a method to retrieve the successors and the predecessors of basic blocks. Edges in CFGs are labeled to distinguish between the True and False branches of a conditional jump. Furthermore, BISM provides an option to display the CFGs of methods before and after instrumentation, which provides developers with visual assistance for analysis and insights on how to instrument the code and optimize it.
Compatibility with ASM BISM uses ASM extensively and relays all its generated class representations within the static context objects. Furthermore, it allows inserting raw bytecode instructions by using the ASM data types. When inserting instructions, it is the user responsibility to write a code free from errors. If the user unintentionally inserts faulty instructions, the instrumentation may fail. The ability to insert ASM instructions provides highly expressive instrumentation capabilities, especially when it comes to inlining the monitor code into the base program, but comes with the risk of producing unwanted behavior.

Instrumentation modes BISM can run in two modes: build-time and load-time. In build-time, BISM acts as a standalone application capable of instrumenting all the compiled classes and methods.\(^1\) In load-time, BISM acts as an agent (using JVM instrumentation capability\(^2\)) that intercepts all classes loaded by the JVM and instruments before the linking phase. The load-time mode permits to instrument additional classes, including classes from the Java class library that are flagged as modifiable.\(^3\) Instrumentation modes are complementary. BISM produces a new statically instrumented standalone program in build-time mode, whereas, in the load-time mode, BISM acts as an interface between the program and the JVM (keeping the base program unmodified).

Portability and ease of use BISM is a lightweight tool written in Java and fitting in a single jar of less than 1Mo. It is hardware-agnostic and only relies on the presence of a JVM in the host software. The user only needs to add the jar to the classpath (Java Runtime Environment variables) to compile new custom transformers. The tool has been successfully tested on various operating systems and even embedded devices such as the Raspberry Pi.

3 BISM language

In this section, we present the language of BISM. The language allows the user to select joinpoints (points in the program execution), retrieve relevant context information, and inject advice (i.e., extra code) that can extract information from these points or alter the behavior of the program.

Instrumentation in BISM is specified in Java classes named transformers. BISM language provides selectors (Sect. 3.1) to select joinpoints of interest, static and dynamic context objects (Sect. 3.2 and Sect. 3.3) which retrieve relevant information from these points, and advice methods (Sect. 3.4) to specify the code to be injected into the base program.

3.1 Selectors

Selectors provide a mechanism to select joinpoints and specify the advice. They are implementable methods where the user writes the instrumentation logic. BISM provides a fixed set of selectors classified into four categories: instruction, basic block, method, and meta selectors. We list below the set of available selectors and specify the execution they capture.

**Instruction** BISM provides instruction-related selectors:
- **BeforeInstruction** captures the execution before a bytecode instruction.
- **AfterInstruction** captures the execution after a bytecode instruction. If the instruction is the exit point, i.e. the last instruction of a basic block, it behaves the same way as the **BeforeInstruction** selector; that is it captures the execution before that last instruction.
- **BeforeMethodCall** captures the execution just before a method call instruction and after loading all needed values on the stack.
- **AfterMethodCall** captures the execution immediately after a method call instruction and before storing the return value of the method call, if any, from the stack into a variable.

**Basic block** In addition to the previous selectors, BISM provides basic block-related selectors that ease capturing control-flow-related execution points:
- **OnBasicBlockEnter** captures the execution when entering the block, before the first real instruction.\(^4\)
- **OnBasicBlockExit** captures the execution after the last instruction of a basic block; except when the last instruction is a JUMP/RETURN/THROW instruction, then it captures the execution before that last instruction.
- **OnTrueBranchEnter** captures the execution on the entry of a successor block after a conditional jump on True evaluation.
- **OnFalseBranchEnter** captures the execution on the entry of a successor block after a conditional jump on False evaluation.

**Method** BISM also provides two method-related selectors:
- **OnMethodEnter** captures the execution on a method entry.
- **OnMethodExit** captures the execution on all exit blocks of a method before the return instruction.

---

\(^1\) Excluding the native and abstract methods, as they do not have bytecode representation.

\(^2\) The java.lang.instrument package.

\(^3\) The modifiable flag keeps certain core classes outside the scope of instrumentation. To the best of our knowledge, there is no exhaustive list of classes with the before-mentioned flag.

\(^4\) Real instructions are instructions that actually get executed, as opposed to some special Java bytecode instructions such as labels or line number instructions.
Meta selectors Finally, BISM provides two class-related meta-selectors: OnClassEnter and OnClassExit. These selectors do not capture execution points but can be used for introductions, such as adding new members to a class. They have no semantics for the execution, but are instead related to BISM execution. Selector OnClassEnter is invoked when a class is loaded and OnClassExit after all methods have been instrumented. They are not related to the static {...} block, which is a function in Java classes. They can also be used to optionally initialize and finalize the transformer execution for each instrumented class.

The order at which selectors are visited when applying a transformer is depicted in Fig. 2. Knowing this traversal flow helps the developer know in which order the advice weaving happens.

3.2 Static context

Static context objects provide access to relevant static information for captured joinpoints in selectors. Each selector has a specific static context object based on its category. These objects can be used to retrieve information about bytecode instructions, method calls, basic blocks, methods, and classes. BISM performs static analysis on the base program and provides additional control-flow-related static information, such as basic block successors and predecessors. The rich set of context information allows the user to have an expressive joinpoint selection mechanism from within selectors. Unlike AspectJ, BISM does not offer yet regular expressions to select joinpoints, but from the context objects one can retrieve the method signature and therefore make the selection manually. It is even possible to be more selective, as BISM offers directly the static context of each bytecode instruction, which is not accessible in AspectJ. The static context hierarchy and accessibility are summarized in Fig. 3. Each context provides access to the corresponding ASM node object and is accessible from the corresponding selector or by traversing the hierarchy bottom-up as demonstrated in Listing 1.

In the following, we detail commonly used properties for each context.

Common properties At each selector, we need to identify the currently instrumented object and its location. Static context objects contain some identifiers:
• a reference to its parent context, named after the parent type;
• multiple string identifiers for class and methods, such as name and signature;
• a unique (method-wise) integer identifier for basic blocks and instructions.

Instruction context  The Instruction context provides relevant information about a single instruction:
• opcode: its opcode in the JVM instruction set;
• next/previous: neighbor instructions in the current basic block, if they exist;
• isBranchingInstruction(): indicator of whether it is a branching instruction (multiple successors). BISM takes care of comparing with the right opcodes and ASM node types.

It is also possible to retrieve information about the stack context at an instruction, as the information is embedded into the class file:
• getBasicValueFrame(): returns a list of all local variables, stack items, and their types at the stack frame before executing the current instruction;
• getSourceValueFrame(): returns a list of all local variables and stack items and their source, i.e., which instruction created/manipulated them.

Method call context  This context object is a special type of Instruction context (only available in MethodCall selectors). In addition to its Instruction context, some specific information is provided, such as the caller method name or the called method class and name.

Basic block context  The BasicBlock context provides information about a basic block and its neighborhood in the CFG:
• blockType: a type to easily identify its role (entry, exit, conditional, or normal block);
• getSuccessor/PredecessorBlocks(): all successors and predecessors of the basic block as per the CFG;
• getTrue/FalseBranch(): the target block after this conditional block evaluates to true (false);
• getFirst/LastRealInstruction(): the first and last executable instructions (not labels) of this block.

Method context  The Method context provides information about the currently visited method:
• name: the name of the method (not fully qualified);
• getEntryBlock/getExitBlocks(): the first and last blocks of the method;
• isAnnotated(String): checks for the existence of an annotation to the method;

```java
class BasicBlockTransformer extends Transformer {
    @Override
    public void onBasicBlockEnter(BasicBlock bb) {
        String blockId = bb.method.className + "." + bb.method.name + "." + bb.id;
        print("Entered block: " + blockId)
    }

    @Override
    public void onBasicBlockExit(BasicBlock bb) {
        String blockId = bb.method.className + "." + bb.method.name + "." + bb.id;
        print("Exited block: " + blockId)
    }
}
```

Listing 2  A transformer for intercepting basic block executions

• some signature information about the method, such as its return type and list of formal arguments.

Class context  The Class context provides the name and ASM node of the currently instrumented class.

In Listing 2, the transformer uses two selectors to intercept all basic block executions (onBasicBlockEnter and onBasicBlockExit). BasicBlock bb is used to get the block id, the method name, and the class name. The advice method print inserts a print invocation in the base program before and after every basic block execution.

3.3 Dynamic contexts

BISM also provides dynamic context objects at selectors to extract joinpoint dynamic information. These objects can access dynamic values from captured joinpoints that are possibly only known during the base program execution. BISM gathers this information from local variables and operand stack, then weaves the necessary code to extract this information. In some cases (e.g., when accessing stack values), BISM might instrument additional local variables to store them for later use. We report here some useful methods, but more are available in the online documentation [32]. For brevity, we omit the return type of the methods, which is always a DynamicValue:

• getThis(): returns a reference to the class owner of the method being instrumented, and null if the class or method is static;
• getName(): returns a reference to the name of the thread executing the method being instrumented;
• getLocalVariable(int): returns a reference to a local variable by index;
• getStackValue(int): returns a reference to a value on the stack by index;
Efficient and expressive bytecode-level instrumentation for Java programs

### 3.4 Advice methods

A user inserts advice into the base program at the captured joinpoints using the advice instrumentation methods. Advice methods allow the user to extract needed static and dynamic information from within joinpoints, also allowing arbitrary bytecode insertion. These methods are invoked within selectors. BISM provides print methods with multiple options to invoke a print command. It also provides (i) invoke methods for static method invocation; (ii) annotate methods for adding annotations to class, methods or fields and (iii) insert methods for inserting bytecode instructions. These methods are compiled by BISM into bytecode instructions and inline at the referenced bytecode location. We list below the advice methods available in BISM.

**Printing on the console** Instrumenting print statements in the base program can be achieved via method print, which permits to write both on the standard and error output of the base program. These methods take either static values or dynamic values retrieved in selectors. Listing 2 shows an example of using one of the print helper methods to instrument the base program to print the basic block constructed id.

**Invoking static methods** Invoking external static methods can be achieved using the advice method invoke. An object of type StaticInvocation should be constructed and provided with the external class name, the method name, and parameters. Listing 3 depicts a transformer that instruments the base program to call the external static method iteratorCreation. The StaticInvocation constructor takes the class and method names as input. Parameters can be added using addParameter(). It supports either DynamicValue type or any primitive type in Java, including String type (any other type will be ignored). After that, invoke weaves the method call in the base program.

**Annotating the bytecode** Annotating elements of a class file allows the user to add some meta-data to the code. It is possible to use the BISM advice annotate(...) to add runtime-visible annotations to either class, fields, or methods. These annotations can be accessed via Java reflection, and they provide extra information for the input program or some third-party API.

**Raw bytecode instruction insertion** Inserting raw bytecode instructions can be achieved with insert methods. When used, it is the developer’s responsibility to write correct instructions that respect the JVM static and structural constraints. Errors can be introduced by ignoring the stack requirements and altering local variables. For Java 8 and above programs, using the insert methods to push new values on the stack or create local variables requires modifying the maxStack and maxLocals values. All static contexts give access to the needed ASM object MethodNode to increment the values maxLocals and maxStack from within the joinpoint.
3.5 Instrumentation scoping

BISM provides many configuration features, such as limiting the scope of the instrumentation or passing arguments to the transformers to modify their behavior. For example, the scope global argument permits matching classes and methods by their names. Specifying \texttt{scope=java.util.List.*, java.util.Iterator.next} will instrument all methods in the \texttt{List} class and only the next method in the \texttt{Iterator} class. Moreover, static context objects can also be used to limit the scope of instrumentation from inside selectors; they can provide more precise scoping information, demonstrated in Listing 3. It is recommended using the scope argument when possible to avoid analyzing unwanted classes, enhancing instrumentation performance.

3.6 User configuration

To favor usability, BISM execution accepts arguments both from the command line (which has higher priority) or through a configuration file. Configurable settings such as printing the CFG files and dumping the instrumented bytecode can be specified. The configuration file is more expressive as it also permits passing arguments to transformers. A transformer may need arguments to modify its internal behavior, e.g., a flag for logging.

4 Instrumentation model

In this section, we introduce the instrumentation model implemented in BISM. We show the correspondence between joinpoints captured by selectors and bytecode regions (i.e., lexical points) in the program given by shadows. We then introduce the equivalence between shadows, which is important when composing transformers (Sect. 5).

4.1 Overview

A \textit{joinpoint} in BISM is essentially a configuration of the base program traversed during its execution. A joinpoint consists of static and dynamic context information. The static context of a joinpoint is defined by a lexical part in the source code. The dynamic context is made of runtime information available in the stack and memory. Depending on the instrumentation task, the user may be interested in some of those joinpoints generated by the program execution. It is at these joinpoints, the user-specified advice is executed.

When instrumenting a program, we inject into its code extra instructions at exact bytecode regions. The user implements selectors (Sect. 3.1) to select joinpoints using lexical conditions that mark such bytecode regions. Lexical conditions are given by shadows. A shadow is a pair defined by a lexical element (instruction, method, basic block) identifier and a direction. Shadows are bytecode instructions along with the direction \texttt{(before and after)}, or basic blocks and methods along with the \texttt{enter} or \texttt{exit} direction. A shadow refers to one bytecode region, and a bytecode region can be referred to by multiple shadows. The joinpoints selection in BISM is statically determinable. That is, determining whether a joinpoint is selected is statically decidable.

4.2 Shadows

Shadows are constructs used to mark the bytecode regions in the base program. BISM internally operates on shadows to extract static information and delimit the regions where the advice is woven. Shadows are pairs consisting of bytecode instructions along with a specified direction \texttt{(before and after)}, or basic blocks and methods along with an \texttt{enter} or \texttt{exit} direction.

Let \texttt{Methods} be the set of all methods that have bytecode representation, \texttt{Blocks} be the set of all basic blocks, and \texttt{Instrs} be the set of all bytecode instructions in the base program. Then, \texttt{Shadows} represents the set of all shadows that BISM identifies in a program.

\[
\text{Shadows} = (\{(\text{before}, \text{after}) \times \text{Instrs}\} \cup (\{(\text{enter}, \text{exit}) \times \text{Blocks}\} \cup (\{(\text{enter}, \text{exit}) \times \text{Methods}\})
\]

For a method \( m \in \text{Methods}, m.\text{blocks} \subseteq \text{Blocks} \) denote the basic blocks in the CFG of \( m \), and \( m.\text{instrs} \subseteq \text{Instrs} \) denote the indexed list of instructions in \( m \).

\textbf{Definition 1 (Method Shadows)}

\( \text{Shadows}_m \) denotes the set of all shadows in method \( m \).

\[
\text{Shadows}_m = (\{(\text{before}, \text{after}) \times m.\text{instrs}\} \cup (\{(\text{enter}, \text{exit}) \times m.\text{blocks}\} \cup \{(\text{enter}, m), (\text{exit}, m)\})
\]

The shadows of a method are restricted to its instructions and basic blocks. We give an example of the shadows in a method as identified by BISM.

\textbf{Example 1 (Method shadows)}

Listing 4 contains a method \( m \) that creates a \texttt{List} \( l \) with an associated \texttt{Iterator} \( i \). The method checks if \( i.\text{hasNext()} \) and calls \( i.\text{next()} \). Listing 5 shows the (simplified) bytecode for method \( m \) in black font. We show some of the shadows identified by BISM, highlighted in the colors blue, red, and olive green. We have two shadows for the method entry and exit points. Two shadows for each basic block (the if-statement results in having three basic blocks),
Efficient and expressive bytecode-level instrumentation for Java programs

Listing 4  A method calling an Iterator

```java
public void m() {
    //Initialize a list of strings
    List<String> l = new ArrayList<>();
    l.add("A");
    //Create iterator
    Iterator<String> i = l.iterator();
    //Call next if iterator has next
    if (i.hasNext())
        i.next();
    System.out.print("done");
}
```

Listing 5  Bytecode and associated shadows for the method in Listing 4

and for each instruction, two shadows to delimit the region before it and after it (we omitted many of the instruction shadows for brevity).

4.3 Selectors matching shadows

Each BISM selector is defined to match a specific subset of the program shadows. Selectors `OnMethodEnter` and `OnMethodExit`, respectively, match the shadows \(<\text{enter}, m>\) and \(<\text{exit}, m>\), for each method \(m \in \text{Methods}\). Selectors `OnBasicBlockEnter` and `OnBasicBlockExit`, respectively, match the shadows \(<\text{enter}, b>\) and \(<\text{exit}, b>\) for each basic block \(b \in \text{Blocks}\). Selectors `OnTrueBranch` and `OnFalseBranch` match special instances of the shadows \(<\text{enter}, b>\) where basic block \(b\) has a predecessor block ending with a conditional jump. Selectors `BeforeInstruction` and `AfterInstruction`, respectively, match the shadows \(<\text{before}, i>\) and \(<\text{after}, i>\) for each instruction \(i \in \text{Instrs}\). Selectors `BeforeMethodCall` and `AfterMethodCall`, respectively, match the shadows \(<\text{before}, i>\) and \(<\text{after}, i>\) for each instruction \(i \in \text{Instrs}\) that is a method invocation instruction.

Example 2 (Selectors matching shadows)

We show the shadows matched by BISM selectors when applying different transformers to the Java program from Example 1. Applying the transformer from Listing 2, the selectors `OnBasicBlockEnter` and `OnBasicBlockExit` match the shadows highlighted in blue in Listing 5. The method has three basic blocks because of the if-statement. These shadows mark the regions where BISM weaves the code to print the basic block id. Applying the transformer from Listing 3, the selector `afterMethodCall` matches the shadow highlighted in the color red in Listing 5. This shadow marks the region where BISM weaves the code to call the monitor `IteratorMonitor.iteratorCreation()`.

Remark 1 (Shadows, traversal, and static analysis)

One of the advantages of the shadows identified by BISM is that they can be used as a traversal strategy for the base program. The traversal strategy can be seen in Fig. 2, where selectors play the role of visitor methods. This allows the user to implement compile-time static analyzers, as we will see in Sect. 6, even if there is no need to instrument the program and insert advice. As such, BISM provides the basic bricks for future implementation of static analysis in combination with runtime verification.

4.4 Equivalence between shadows

A bytecode region in the program can be targeted by different selectors. Since shadows mark the bytecode regions, we define the notion of equivalence between shadows which allows us to detect when selectors target the same bytecode regions.

Definition 2 (Equivalence Relation over Shadows)

The equivalence relation over shadows in a method \(m\) is denoted by \(\equiv_s^m\) and defined as follows:

\[
\equiv_s^m \subseteq \text{Shadows}_m \times \text{Shadows}_m
\]
Springer

applying multiple transformers in a single run as base program in the order specified by the user. We refer to a single run. The transformers are applied sequentially to the program bytecode regions (Sect. 5.2) and how BISM can help (Sect. 5.3 and Sect. 5.4).

Example 3 (Equivalent shadows in a method)
Fig. 4, depicts the CFG of a method \( m \) with 4 basic blocks \( b_1, b_2, b_3, b_4 \) where \( b_1 \) is the entry block, \( b_2 \) and \( b_4 \) are both exit blocks. In basic block \( b_2 \), we show two consecutive instructions \( i \) and \( j \). In basic block \( b_3 \), we show instruction \( k \) as the first instruction in the block and instruction \( l \) as the last instruction. The filled grey boxes in the figure illustrate the equivalent shadows, numbered as their corresponding line in Definition 2. From (1), we have \( \langle \text{enter}, m \rangle \equiv^m \langle \text{enter}, b_1 \rangle \). From (2), we have \( \langle \text{exit}, b_4 \rangle \equiv^m \langle \text{exit}, b_2 \rangle \). From (3), we have \( \langle \text{enter}, b_1 \rangle \equiv^m \langle \text{exit}, m \rangle \). From (4), we have \( \langle \text{after}, l \rangle \equiv^m \langle \text{exit}, b \rangle \). From (5), we have \( \langle \text{after}, i \rangle \equiv^m \langle \text{before}, j \rangle \).

5 Transformer composition

BISM allows the user to apply more than one transformer in a single run. The transformers are applied sequentially to the base program in the order specified by the user. We refer to applying multiple transformers in a single run as transformer composition. In this section, we discuss the motivation for composing transformers (Sect. 5.1) and address some concerns that may arise when multiple transformers target the exact program bytecode regions (Sect. 5.2) and how BISM can help (Sect. 5.3 and Sect. 5.4).

More than one pass In many cases, transformations might require multiple passes on the same class. Since BISM can be used to implement static analyzers, this enables plenty of scenarios where static analysis can be leveraged in combination with runtime verification. In such cases, a transformer can be implemented to perform the analysis before the transformer responsible for instrumenting the code for monitoring. Another example is assuming that we are implementing a simple obfuscator that randomly changes the names of all
methods in a program. In this case, one pass is not enough. We need one pass to map the original names to the obfuscated names and then another pass to change the classes and method names.

**Modularity of transformers** At the core of aspect-oriented programming is achieving modularity to the cross-cutting concerns of an application. Hence we encourage separating transformers based on their functionality. This allows different team members to implement different transformers separately, where a single transformer should logically handle one concern. Let us say we want to instrument a program to monitor at runtime safety properties such as isNext, UnsafeIterator and SafeSyncMap (see Sect. 8.3). Implementing a single transformer for each property is more readable and favors reuse.

### 5.2 Transformer collision

Transformers impose new “aspects” into the base program through inserting advice. When two transformers insert advice that targets the same program bytecode regions, we say that the two transformers collide. BISM detects and reports transformer collisions, which makes the composition more transparent to the user.

BISM detects collision after weaving the advice of multiple transformers into the base program. Recall Definition 1 of the shadows of a method. Let $\text{Shadows}_m^t$ represent all the shadows used by BISM to insert the advice for transformer $t$, in a method $m$, we have:

$$\text{Shadows}_m^t \subseteq \text{Shadows}_m$$

To detect collision in a method, we check whether two transformers insert advice at equivalent shadows (Sect. 4.4).

**Definition 3 (Transformer Collision)**

Transformer $t$ collides with transformer $t'$ in method $m$, if

$$\exists s \in \text{Shadows}_m^t, \exists s' \in \text{Shadows}_m^{t'} : s \equiv^m s'$$

Notice that the collision between transformers is symmetric, which means that the order of applying two transformers is irrelevant to detect collision. Also, collision is reflexive, which means that collision is also detected when applying the same transformer twice.

**Collision report** To detect a collision, we compute equivalent shadows used for instrumentation by transformers. BISM records the used shadows after weaving each transformer and reports all collisions after each run. The report shows the exact locations of collision along with the colliding transformers to the user.

```java
public class Logging extends Transformer {
    public void onMethodEnter(...) {
        // Log methodName + Enter
    }
    public void onMethodExit(...) {
        // Log methodName + Exit
    }
}
public class Timer extends Transformer {
    public void onMethodEnter(...) {
        // Initialize a timer object
    }
    public void onMethodExit(...) {
        // Log time elapsed + methodName
    }
}
```

**Listing 6** Logging and Timer transformers (written in pseudo-code)

Several concerns may arise from collisions, such as determining the order of execution and the visibility among aspects. We discuss these problems in the rest of this section.

### 5.3 Order matters

BISM applies the transformations in the order specified by the user as input in the command line or configuration file. One issue that may arise is that applying the transformations in a different order exhibits different behavior in the final instrumented program. Let us look at the following example.

**Example 4 (Order matters)**

Listing 6, demonstrates two transformers: Logging, which is used to log enters and exits to all methods, and Timer that profiles the time needed for each method to execute. We can see that the joinpoints captured by onMethodEnter and onMethodExit are common between both transformers. Hence the advice is inserted at the same bytecode regions, and we have a collision. Applying transformer Logging before Timer results in the timer calculating the original method computation in the base program and the time needed by the logger. We may want to give precedence to the timer to calculate the time without the logging operations; hence logging will wrap the base program and the timer operations.

In other cases, the execution order is not important for us, even if there is a collision between two transformers. In Sect. 8.3, we implement multiple transformers to instrument and monitor different safety properties. In this case, changing the order of the transformations does not produce any observable semantic difference in the final instrumented program. In general, the user is encouraged to check the collision report (Sect. 5.2) and manually change the order of transformers if needed.
5.4 Controlling visibility

When composing transformers, each transformer introduces a new set of instructions to the base program. The newly added instructions are then part of the base program and become visible to the second transformer to target. In many cases, we may want to hide these newly instrumented instructions in a composition. BISM provides the attribute @Hidden that can be placed as an annotation on a transformer. When used, the newly added instructions are not intercepted anymore by the selectors by following transformers. A prevalent scenario is to avoid instrumenting previously added code. Let us look at the following example.

**Example 5 (Hidden transformers)**

Listing 7 demonstrates two different transformers: CountMethodCalls which counts the number of method calls, and LogMethodCalls, which logs all method calls. We can see that the joinpoints captured by onMethodCall are shared between both transformers. Hence the advice is inserted at the same bytecode regions and we have a collision. A user might be only interested in logging the method calls of the base program and does not want the logger to log counting calls introduced by LogMethodCalls. Alternatively, the user might be interested in counting the method calls of the base program and not the log calls introduced by CountMethodCalls. Adding the @Hidden attribute to the transformers hides the newly added instructions from other transformers in a composition.

**Hidden instructions** BISM also allows transformers to hide arbitrary instructions of the base program from other transformers by providing a mechanism to mark instructions as hidden. When an instruction is marked as hidden, it is excluded from Shadows\_m and thus not exposed to selectors. Hence it will not be intercepted by the transformers that follow. This feature can be used for optimizing instrumentation by having one transformer implement a static analyzer that hides particular instructions from the instrumentation transformer.

In general, to avoid instrumenting previously added advice, the user is encouraged to check the collision report (Sect. 5.2) and use the @Hidden when needed.

6 Some example use cases

In this section, we show the versatility of BISM by demonstrating some use cases in static and dynamic analysis of programs, namely the runtime verification, logging, dynamic profiling, mutation, and code analysis of programs.

6.1 Good Java practices

This section demonstrates how to use BISM to instrument the code for monitoring classical runtime verification properties about good Java practices. We do not discuss the monitor and assume that it is implemented in a separate library.

6.1.1 HasNext property

We consider the HasNext property on iterators, which specifies that the hasNext() method should be called and return true before calling the next() method on an iterator. Listing 8 shows a BISM transformer for monitoring the property. We use the method call joinpoints and filter for invocations of hasNext() and next() on iterator objects using the mc object, which exposes static context from the captured method call. We use the dynamic context object dc to retrieve the object receiving the method call, in this case, the Iterator instance. The getMethodReceiver (explained in Sect. 3.3) retrieves the iterator instance by loading it from the stack into a local variable returning a reference to it in a DynamicValue object. We assume that a monitor is implemented in a separate class with two static methods hasNext() and next().

We invoke each method, respectively passing the iterator instance to the monitor using the BISM invocation helper method StaticInvocation.

6.1.2 Safe locking

The SafeUnlock property specifies that the number of acquires and releases of a (reentrant) Lock class are matched within a given method call. In Listing 9, the transformer captures lock and unlock operations in a method and extracts dynamic context such as the thread name, the lock object, the calling object instance. A monitor is implemented in a separate class with the two static methods lockOperation() and unlockOperation(). We invoke each method, respectively passing the extracted values to the monitor.
6.2 Logging

Listing 8 HasNext instrumentation

```java
public void afterMethodCall(MethodCall mc,
   MethodCallDynamicContext dc) {
   if (mc.methodName.contains("hasNext") &&
      mc.methodOwner.contains("Iterator")) {
      DynamicValue iterator =
         dc.getMethodReceiver(mc); //Instance of the iterator
      DynamicValue result = dc.getMethodResult(mc);
      StaticInvocation sti = new
         StaticInvocation("Monitor", "hasNext");
      sti.addParameter(iterator);
      sti.addParameter(result);
      invoke(sti);
   }
   public void afterMethodCall(MethodCall mc,
   MethodCallDynamicContext dc) {
   if (mc.methodName.contains("next") &&
      mc.methodOwner.contains("Iterator")) {
      DynamicValue iterator =
         dc.getMethodReceiver(mc);
      StaticInvocation sti = new
         StaticInvocation("Monitor", "next");
      sti.addParameter(iterator);
      invoke(sti);
   }
}
```

Listing 9 SafeLock instrumentation

```java
0Override
public void beforeMethodCall(MethodCall mc,
   MethodCallDynamicContext dc) {
   if (mc.methodName.equals("lock") &&
      mc.methodOwner.contains("Lock")) {
      DynamicValue threadName =
         dc.getThreadName(mc);
      DynamicValue lockObject =
         dc.getMethodReceiver(mc); //Instance of the Lock
      DynamicValue _this = dc.getThis(mc);
      String currentMethod = mc.ins.methodName;
      StaticInvocation sti = new
         StaticInvocation("Monitor", "lockOperation");
      sti.addParameter(threadName);
      sti.addParameter(lockObject);
      sti.addParameter(_this);
      sti.addParameter(currentMethod);
      invoke(sti);
   }
0Override
public void afterMethodCall(MethodCall mc,
   MethodCallDynamicContext dc) {
   if (mc.methodName.equals("unlock") &&
      mc.methodOwner.contains("Lock")) {
      DynamicValue threadName =
         dc.getThreadName(mc);
      DynamicValue lockObject =
         dc.getMethodReceiver(mc); //Instance of the Lock
      DynamicValue _this = dc.getThis(mc);
      String currentMethod = mc.ins.methodName;
      StaticInvocation sti = new
         StaticInvocation("Monitor", "unlockOperation");
      sti.addParameter(threadName);
      sti.addParameter(lockObject);
      sti.addParameter(_this);
      sti.addParameter(currentMethod);
      invoke(sti);
   }
}
```

6.3 Dynamic profiling

We demonstrate how to implement dynamic profiling with BISM. We collect dynamic context from a running program, including the number of method invocations, runtime types of method arguments (Sect. 6.3.1), number of allocated objects (Sect. 6.3.2), and return types (Sect. 6.3.3). We do not
focus on implementing the profiler tool, but only on how to extract context using BISM.

6.3.1 Call graph

We consider the dynamic call graph of a program, which represents the calling relationship between methods in program execution. For each method call in an execution, we are interested in extracting runtime information from the caller and callee methods. Listing 11 shows the code of a transformer that instruments to extract the caller and callee classes and method names along with their runtime arguments for each method call. The arguments of the caller and callee are extracted using the dynamic context method `dc.getMethodArgs()`. We instrument two synthetic local arrays in the base program to store the extracted values locally in the method. For the caller, the arguments are retrieved once at method enter to avoid repeating the argument extraction for each invocation by the caller. At `onMethodEnter`, calling `dc.getMethodArgs()` will retrieve the needed values from the local variables of the method. As for the callee, the `dc.getMethodArgs()` will retrieve the arguments directly from the stack. Then, before each method call, an invocation to the profiler method `callGraph` is instrumented, passing the static and dynamic information.

6.3.2 Object allocation

Object allocation is an important metric in dynamic profiling that allows the user to know the number of created objects in the program and estimate the used memory. Listing 12 shows a transformer that instruments to capture allocated objects and arrays in a program. We use the `beforeInstruction` joinpoint and filter for all `NEW` opcodes. To extract the type of the created object, we use the access granted by BISM to the ASM instruction node object and get more details from the bytecode instruction. The extracted static information is then passed to the profiler by invoking its appropriate method.

6.3.3 Return types

Listing 13 shows how to extract return types from methods. We use the `afterMethodCall` joinpoint and filter using the static context provided `mc.returns`, where return is a boolean flag indicating if the method has a return type in its signature. Then, we extract the return result into the dynamic value object `dv`. After that, an invocation to the profiler is instrumented, which passes the needed information. We choose to box the return value for a more generic implementation.

6.4 Mutation of programs

We consider software testing and, more particularly, mutation testing (see [22] for a survey). Mutation testing aims to ensure software quality by checking that slightly modified versions of a program (i.e., mutants) will not pass the same tests as the original. Mutants emulate the programs that would be obtained as the result of programmers’ mistakes.
Efficient and expressive bytecode-level instrumentation for Java programs

Listing 12 Profiling object allocation

```java
@Override
public void beforeInstruction/Instruction
    ,...) {
  //Object creation opcodes
  if (ins.opcode == Opcodes.NEW
      || ins.opcode == Opcodes.NEWARRAY
      || ins.opcode == Opcodes.ANEGARRAY
      || ins.opcode == Opcodes.MULTIANEWARRAY) {
  TypeInsnNode instruction = (TypeInsnNode) ins.node;
  //Invoke profiler
  StaticInvocation sti = new
      StaticInvocation("Profiler",
          "allocation");
  sti.addParameter(ins.method.fullName);
  sti.addParameter(ins.opcode);
  sti.addParameter(instruction.desc);
  invoke(sti);
}
}
```

Listing 13 Profiling return types

```java
@Override
public void afterMethodCall/Instruction
    (MethodCall mc,
     MethodCallDynamicContext dc){
  //If a method returns
  if (mc.returns) {
    //Get the result
    DynamicValue dv = dc.getMethodResult(mc);
    //Invoke profiler
    StaticInvocation sti = new
        StaticInvocation("Profiler",
            "returnTypes");
    sti.addParameter(caller);
    sti.addParameter(mc.fullName);
    sti.addBoxedParameter(dv);
    invoke(sti);
  }
}
```

There are various types of mutations of various complexity levels [10, 29]. We consider the following types of often-occurring mutations:

- Value mutations, which change variable values into the program or return values.
- Operator mutations, which change the logical or arithmetical operators used across the program.
- Statement mutations, which change complex constructions, like method calls or even the CFG of the program.

In the following, we define an example mutator for each type of mutation, i.e., a transformer producing such mutations.

6.4.1 Return mutator: value mutation

The mutator in Listing 14 emulates the fact that a default return value has been forgotten in the program. Hence, the target method always returns the same fixed value instead of the normally computed one. For this, the mutator uses the onMethodExit joinpoint and detects whether the parameter method returns a value using the method type. In such a case, the mutator removes the value from the stack. Then, a fixed value (here 0 for integers) is pushed onto the stack to be returned.

6.4.2 Instruction mutator: operator mutation

The mutator in Listing 15 performs some replacements on a specified set of instructions. The mutator is generic and relies on some abstract methods. A replacement instruction can either be randomly chosen or obtained using a user-defined mapping between instructions. To do this for an instruction, the mutators check whether the instruction is in its scope and if so, it replaces it.

We present two instances of the operator mutator, which are obtained by implementing the abstract methods.

- The mutator in Listing 16 targets conditional operators, which are detected as conditional jump instructions. Another comparison operator replaces conditional operators without changing their destination.
- The mutator in Listing 17 targets binary arithmetic operators on integers. Arithmetic operators are replaced either by a random operator or the complementary one (− and +, & and | for bitstring operators...).
beforeInstruction(Instruction ins, ...){
    if (isCovered(ins)){
        remove(ins);
        if (negate)
            insert(negate(ins));
        else
            insert(random(ins));
    }
    //Check whether a particular instruction is covered (type, position, ...)
    abstract boolean isCovered(Instruction);
    //Choose a random operation (compatible in term of type, arg count ...)
    abstract AbstractInsnNode random(Instruction);
    //Negate the opcode of a given instruction when applicable
    abstract AbstractInsnNode negate(Instruction);
}

Listing 15  Generic instruction mutator

//If it is a conditional
isCovered(Instruction ins) {
    return ins.isConditionalJump();
}

//Choose a random if which is compatible in term of type and arg count
random(Instruction ins) {
    if (insIf.opcode >= Opcodes.IFEQ &&
        insIf.opcode <= Opcodes.IFNE)
        return new
            JumpInsnNode(randomIFBetween(IFEQ, IFNE),
            insIf.node.label);

    //Negate the opcode of a given if
    negate(Instruction ins) {
        if (ins.opcode == Opcodes.IFNULL ||
            ins.opcode == Opcodes.IFNONNULL)
            return new
                JumpInsnNode(ins.opcode +1,
                ins.node.label);
        else
            return new
                JumpInsnNode(ins.opcode -1,
                ins.node.label);
}

Listing 16  Decision mutator

6.4.3 Void call mutator: statement mutation

The mutator in Listing 18 removes calls to methods with the void return type. For this, whenever there is a call to such a method, the transformer unloads its parameters from the stack and removes the INVOKEEX opcode. To check for return types and unload the parameters differently regarding their sizes, the transformer iterates through the method descriptor available through the static context attribute mc.methodnode.desc.

```
final List<Integer> I2Opcodes = List(  
    Opcodes.IADD, Opcodes.ISUB,  
    Opcodes.IMUL, Opcodes.IDIV,...);  

isCovered(Instruction ins){
    //All double int operand arithmetic instructions
    return I2Opcodes.contains(ins.opcode);
}

random(Instruction ins){
    return new InsnNode(I2Opcodes.get(  
        Math.random() * I2Opcodes.size()));
}

negate(Instruction ins){
    return new InsnNode(ins.opcode + (  
        I2Opcodes.indexOf(ins.opcode) % 2 == 0 ?  
        1 : -1));
}

Listing 17  Arithmetic mutator

beforeMethodCall(MethodCall mc, ...){
    if (getReturnType(mc.methodnode) != VOID_TYPE)
        return;
    //Pop each argument, respecting its size
    for (var arg: getArgumentTypes(mc.methodnode))
        insert(new InsnNode(arg.getSize() == 1 ?
            POP : POP2);
    remove(mc.ins);
}

Listing 18  Void call mutator
```

6.5 Code analysis of programs

We consider the analysis of program codes along quality metrics on class files. Software quality is a classic concern in software engineering. Measuring software quality is instrumental in ensuring several properties such as low technical debt, upgradeable software, and secure coding. In [21, 23], white-box (i.e., based on source code) analysis metrics are defined to measure quality, understandability, and maintainability. The higher level of abstraction and the updatability of the source code (access to the documentation, comments, fully structured ...) are incentives for defining code analysis techniques on source code. As such, there is a lack of tools to compute quality metrics on the bytecode.

BISM permits accessing and computing many valuable properties that can be used to compute standard metrics relying on the CFG of methods, the number of variables and

5 The descriptor is a string representing a type, for a method it permits to access the return and argument types.
Efficient and expressive bytecode-level instrumentation for Java programs

Listing 19  McCabe cyclomatic complexity

method calls, and the program instructions. This makes such analysis possible on legacy software.

While BISM does not provide access to the source code, nor to some classical metrics like Lines Of Code or NPATH complexity, it still provides essential static information. Next, we show how to compute the following software quality metrics: McCabe cyclomatic complexity, ABC Metric, and the count of unused variables.

McCabe complexity  The McCabe cyclomatic complexity [28] is defined as the maximum number of independent paths in a CFG. For a CFG $G$, it is easily computable by: $V(G) = |Edges_G| - |Nodes_G| + 2$. In Listing 19, the transformer uses the computed CFG to count the number of conditional edges inside it.

ABC complexity  To compute the ABC complexity, we only need to classify instructions and basic blocks. Computing the ABC complexity [18] relies on the capability to distinguish between branching, assignments, and conditional jumps. Listing 20 shows a transformer that computes the complexity using blockType and opcode fields of static contexts.

Unused variables  We consider that a variable is not used in a method if it is never loaded within the method. For this, the transformer in Listing 21 checks whether an instruction is a direct load which takes as parameter a variable index and pushes its value onto the stack. In such a case, the transformer retrieves the index of the variable and sets it as not unused in the mapping implemented by boolean array unusedVars. The check is run on all instructions, and the variables which have never been loaded on the stack are declared unused.

7 BISM implementation

In this section, we provide some details about BISM implementation. BISM is implemented in Java using about 7,000 LOC and 55 classes distributed in separate modules. It uses ASM under the hood for bytecode parsing, analysis, and weaving. BISM is provided as a runnable JAR file that requires no installation from the user except for having Java 8 or above installed. It can run in two modes: build-time mode, as a standalone application to statically instrument a program, and load-time mode, attached to a program as a
Java agent. Figure 5 shows a detailed view of the internal workflow.

(1) **User Input.** In build-time mode, arguments consist of a base program bytecode (.class or .jar) to be instrumented and a list of transformers that specifies the instrumentation logic. In load-time mode, only the transformers are passed as arguments and all classes loaded by the JVM are instrumented. BISM provides several built-in transformers that can be directly used. Moreover, users can specify various runtime arguments to BISM or even the transformers from the console or through a configuration file.

(2) **Parse Bytecode.** For each class in the base program, BISM uses ASM to parse the bytecode and generate a tree object containing all the class details, such as fields, methods, and instructions. The following three steps will be performed on each class for every transformer specified in a run.

(3) **Build CFG.** BISM constructs the CFGs for all methods in the target class. If the transformer utilizes control-flow joinpoints (onTrueBranch and onFalseBranch), BISM eliminates all critical edges from the CFGs to avoid instrumentation errors. This is done by inserting empty basic blocks in the middle of critical edges, which is only applied if used while keeping copies of the original CFGs. Also, if the transformer uses joinpoint onMethodExit, all the exit blocks (which terminate with a return opcode) are merged into a single one to avoid duplication and errors. This is done by adding a new block that contains a return of a suitable type; then, all other returns are replaced by unconditional jumps to the added one. Moreover, if the users opted for the visualizer, the CFGs are printed into HTML files on the disk.

(4) **Generate Shadows and Context Objects.** BISM iterates over the target classes to identify all shadows utilizing the created CFGs. The relevant static and dynamic context objects are created and initialized using the static information available and BISM analysis at each shadow.

(5) **Transformer Weaving.** The transformer is notified of each shadow and passed the static and dynamic objects. The weaving loop is illustrated in Fig. 2. BISM evaluates the transformations applied by a transformer using the advice methods. After that, it accordingly weaves the necessary bytecode instructions into the target class.

(6) **Output.** The instrumented bytecode is then output back as a .class file in build-time mode or passed as raw bytes to the JVM in load-time mode. In case of instrumentation errors, e.g., due to adding manual faulty ASM instructions by the user, a weaving error is emitted. If the visualizer is enabled, the instrumented CFGs are also printed into HTML files on the disk.

8 Performance evaluation

We report on our performance evaluation of BISM.

**Experiments and used programs.** We compare BISM with DiSL and AspectJ, which are the most popular tools for the monitoring and runtime verification of Java programs. For this, we use three complementary experiments. Additionally, Table 1 shows how the three experiments are complementary to each other.

- The first experiment concerns the implementation of the Advanced Encryption Standard (AES). This experiment shows how BISM can perform inline instrumentation by inserting new bytecode instructions inside the target program to detect test inversion attacks on the application.
- The second experiment concerns a financial transaction system. This experiment shows how BISM can be used to instrument the system to monitor user-provided properties. The financial transaction system is a relatively small application with a low event rate.
- The third experiment concerns the DaCapo benchmark [6]. This experiment shows how BISM can be used to instrument the benchmark and monitor for the good usage of data structures (with classical properties: HasNext, UnsafeIterator, and SafeSyncMap). DaCapo is a large benchmark.

---

6 We use the latest versions of DiSL from https://gitlab.ow2.org/disl/disl and AspectJ Weaver 1.9.4.
classically used when evaluating runtime verification tools as it produces events at a high rate.

For the first experiment, instrumentation is performed at the level of the control-flow graph. For the two other experiments, the instrumentation is performed at the level of method calls to emit events. Note, AspectJ is not capable of instrumenting for inline monitoring of control-flow events, so we do not include it in the first experiment (with AES).

We run our experiments in both of BISM instrumentation modes, namely load-time and build-time. Running an experiment in load-time mode serves to compare the performance when the tools act as an interface between the base program and the virtual machine. Running an experiment in build-time mode serves to compare the performance of the generated instrumented bytecode.

We note that DiSL wraps its instrumentation code with exception handlers. Exception handlers are not necessary for our experiments and have a performance impact. To guarantee fairness, we switched off exception handlers in DiSL.

**Evaluation metrics** We consider three performance metrics: runtime, used memory, and bytecode-size. We are interested in evaluating the instrumentation overhead, that is, the performance degradation caused by instrumentation. For each metric, we use the base program as a baseline. For runtime, we measure the execution time of the instrumented program. For used memory, we measure the used heap and non-heap memory after a forced garbage collection. In load-time mode, we do not measure the used memory in the case of DiSL, because DiSL performs instrumentation on a separate JVM process.

**Evaluation environment** To run the experiments, we use Java JDK 8u251 with 2 GB maximum heap size on an Intel Core i9-9980HK (2.4 GHz. 8 GB RAM) running Ubuntu 20.04 LTS 64-bit. We consider 100 runs and then calculate the mean and the standard deviation.

In what follows, we illustrate how we carried out our experiments and the obtained results.\(^7\)

### 8.1 Advanced encryption standard (AES)

**Experimental setup** We compare BISM with DiSL in a scenario using inline monitors. We instrument an external AES implementation to detect test inversions in the control flow of the program execution. The instrumentation deploys inline monitors that duplicate all conditional jumps in their successor blocks to report test inversions. We implement the instrumentation as follows.

8 Extracting stack values can be also alternatively achieved using dynamic context method `getStackValue` and adding new local variables.

9 `OnTrueBranchEnter`, `onFalseBranchEnter`.

---

\(^7\) More details about the experiments and the material needed to reproduce them can be found at https://gitlab.inria.fr/bism/bism-experiments.
Table 1 A comparison between the experiments. LT is for load-time mode, and BT is for build-time mode. A checkmark (✓) indicates that the experiment involves the metric or the feature, whereas a cross mark (✗) indicates that the experiment does not involve the metric or the feature. Term NA abbreviates Not Applicable, and (-DiSL) indicates that the DiSL tool has been excluded.

|                  | Performance Metrics | Instrumentation Level | Bytecode Insertion | Comparison with |
|------------------|---------------------|-----------------------|--------------------|-----------------|
|                  | Runtime             | Used Memory           | Bytcode Size       | AspectJ         | DiSL            |
| AES              | LT                  | ✓                     | ✓                  | ✓               | ✓               |
|                  |                     | (-DiSL)               | NA                 |                 |                 |
|                  | BT                  | ✓                     | ✓                  | ✓               | ✓               |
| Transactions     | LT                  | ✓                     | ✓                  | ✓               | ✓               |
|                  |                     | (-DiSL)               | NA                 |                 |                 |
|                  | BT                  | ✓                     | ✓                  | ✓               | ✓               |
| DaCapo           | LT                  | ✓                     | ✓                  | ✓               | ✓               |
|                  |                     | (-DiSL)               | NA                 |                 |                 |
|                  | BT                  | ✓                     | ✓                  | ✓               | ✓               |

Fig. 7 AES build-time instrumentation

Table 2 Number of emitted events in AES experiment

| Plain-text size (kB) | Events (M) |
|----------------------|------------|
| 2^0                  | 0.9        |
| 2^1                  | 1.8        |
| 2^2                  | 3.6        |
| 2^3                  | 7.3        |
| 2^4                  | 14.9       |
| 2^5                  | 29.5       |
| 2^6                  | 58.5       |
| 2^7                  | 117        |
| 2^8                  | 233        |

Build-time evaluation We replace the original classes of AES with statically instrumented classes of each tool. Figure 7 reports the runtime and used memory in build-time mode depending on plain-text size. BISM shows less overhead than DiSL in both runtime and used memory for all plain-text sizes. Moreover, BISM incurs a relatively small overhead for all plain-text sizes. Table 2 reports the number of generated events (corresponding to conditional jumps) after running the code (in millions).

The bytecode size of the original AES class is 9 kB. After instrumentation, the bytecode size is 10 kB (+11.11%) for BISM, and 128 kB (+1,322%) for DiSL. So, BISM incurs less bytecode-size overhead than DiSL. The significant overhead in DiSL is due to the inability to inline the monitor in bytecode and having to instrument it in Java. We note that it is not straightforward in DiSL to extract control-flow information in Markers, whereas BISM provides this out-of-the-box.

8.2 Financial transaction system

Experimental setup We compare BISM with DiSL and AspectJ in a runtime verification scenario to monitor some properties of a financial transaction system. We use the implementation from CRV-14 [3] to monitor the following properties:

- Property P1: only users based in certain countries can be Silver or Gold users.
• Property P2: the transaction system must be initialized before any user logs in.
• Property P3: no account may end up with a negative balance after being accessed.
• Property P4: an account approved by the administrator may not have the same account number as any other already existing account in the system.
• Property P5: once a user is disabled by the administrator, he or she may not withdraw from an account until being activated again by the administrator.
• Property P6: once greylisted, a user must perform at least three deposits from external before being whitelisted.
• Property P7: no user may request more than 10 new accounts in a single session.
• Property P8: the administrator must reconcile accounts every 1000 external transfers or an aggregate total of one million dollars in external transfers.
• Property P9: a user may not have more than three active sessions at once.
• Property P10: transfers may only be made during an active session (i.e., between a login and logout).

For each property using a set of events, we instrument the financial transaction system to generate those events. Such events mainly correspond on the system to method call with parameters or class field updates. For example, monitoring Property P6 requires the following events: greylistUser(id), depositFromExternal(id) and whitelistUser(id), where id is a unique user identifier. As test suite, we implement a custom set of scenarios that covers all of the above properties, and an external monitor library with stub methods that only count the number of received events. We implement instrumentation as follows:

• In BISM, we use the static context provided at method-call instrumentation selectors\(^{10}\) to filter methods by their names and owners. To access the method calls’ receivers and results, we utilize methods getMethodArgs and getMethodResult available in dynamic contexts. We then use argument processors and dynamic context objects to access dynamic values and pass them to the monitor. The extracted values are then passed to the monitor by invoking its appropriate method.
• In DiSL, we implement custom Markers to capture the needed method calls and use argument processors and dynamic context objects to access dynamic values. We note that it required to create a custom marker for each method call, which resulted in implementing 28 different marker classes.
• In AspectJ, we use the call pointcut, type pattern matching, and joinpoint static information to capture method calls and write custom advices that invoke the monitor.

### Load-time evaluation
Figure 8 reports the runtime and used memory for the considered properties in load-time mode (excluding DiSL in the case of used memory). BISM shows better performance over DiSL and AspectJ for properties P2, P5, P6, P8, and P10, for five properties out of ten. Whereas DiSL shows the best performance for P3 and P4, and AspectJ shows the best performance for properties P1, P7, and P9. The similar results of the tools are due to the fact that each property monitor augments the base program with a small number of advices at limited locations, ranging between two and five advices per property. Hence, the results in load-time mode reflect the execution time of the woven advice more than the instrumentation overhead. Concerning used memory, BISM incurs much lower overhead than AspectJ for all properties.

### Build-time evaluation
We replace the original classes of the scenarios with statically instrumented classes from each tool. Figure 9 reports the runtime and used memory for the considered properties in build-time mode. BISM shows less runtime and used-memory overheads than both DiSL and AspectJ for all properties. Table 3 reports the number of generated events after running the code (in thousands). The bytecode size of the classes of the overall original scenarios is 44 KB. After instrumentation, the bytecode size is 56 KB (\(+27.27\%)\) for BISM, 84 KB (\(+90.9\%)\) for DiSL, and 116 KB (\(+163.63\%)\) for AspectJ. Hence, BISM incurs less bytecode-size overhead than both DiSL and AspectJ.

### 8.3 DaCapo benchmarks

#### Experimental setup
We compare BISM with DiSL and AspectJ in a general runtime verification scenario. We instrument the benchmarks in the DaCapo suite [6] (dacapo-9.12-bach), to monitor for the classical HasNext, UnsafeIterator, and SafeSyncMap properties.\(^{11}\) We only target the packages specific to each benchmark, and do not limit our scope to java.util types; instead, we match freely by type and method name. We implement an external monitor library with stub methods that only count the number of received events. We implement the instrumentation similarly to the second experiment:

• In BISM, we use the static context provided at method-call instrumentation selectors to filter methods.
• In DiSL, we implement custom Markers to capture the needed method calls.

\(^{10}\) beforeMethodCall, afterMethodCall.

\(^{11}\) The HasNext property specifies that a program should always call method hasNext() before calling method next() on an iterator. The UnsafeIterator property specifies that a collection should not be updated when an iterator associated with it is being used. The SafeSyncMap property specifies that a map should not be updated when an iterator associated with it is being used.
In AspectJ, we use the call pointcut, type pattern matching, and joinpoint static information to capture method calls. We choose the following benchmarks: avrora, batik, fop, h2, pmd, sunflow and xalan. For each benchmark, Dacapo provides a small, default and large workload. We choose the default workload, which includes a number of warm-up runs performed internally.

Load-time evaluation Figure 10 reports the runtime for the benchmarks. BISM shows better performance than DiSL and AspectJ for all benchmarks. DiSL shows better performance than AspectJ except for the pmd benchmark. For the pmd benchmark, this is mainly due to the fewer events emitted by AspectJ (see Table 4). We notice that AspectJ captures fewer events in benchmarks batik, fop, pmd, and sunflow. This is due to its inability to instrument synthetic bridge methods, generated by the compiler after type erasure in generic types. BISM also shows less used-memory overhead over AspectJ in all benchmarks. Let us mention that we did not measure the used memory for DiSL, since it performs instrumentation on a separate JVM process.

| Property | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 | P10 |
|----------|----|----|----|----|----|----|----|----|----|-----|
| Events (k) | 10 | 10 | 11 | 33 | 71 | 69 | 125 | 104 | 192 | 154 |

Fig. 8 Financial transaction system load-time instrumentation

Fig. 9 Financial transaction system build-time instrumentation
Fig. 10 DaCapo load-time instrumentation

Build-time evaluation We replace the original classes in the benchmarks with statically instrumented classes from each tool. Figure 11 reports the runtime and used memory of the benchmarks. BISM shows less runtime overhead in all benchmarks, except for batik, where AspectJ emits fewer events. BISM also shows less used-memory overhead, except for sunflow, where AspectJ emits much fewer events.

Table 4 compares the instrumented bytecode. We report the number of classes in scope (Scope) and the instrumented (Instr.), and we measure the bytecode-size overhead (Ovh.) for each tool. We also report the number of generated events after running the code (in millions). BISM and DiSL generate the same number of events, while Aspect (AJ) produces fewer events because of the reasons mentioned above. The results show that BISM incurs less bytecode-size overhead for all benchmarks. We notice that even with exception-handlers turned off, DiSL still wraps a targeted region with try-finally blocks when the @After annotation is used. This guarantees that an event is emitted after a method call, even if an exception is thrown.

Fig. 11 DaCapo build-time instrumentation

8.4 Threats to validity

One of the threats to the validity of our experiments is the non-determinism of the DaCapo benchmarks: avrora, h2, pmd, sunflow, and xalan. We mitigate this by measuring the mean execution times over 100 runs. We also measure dispersion (variance and standard deviation) and report it in the graphs. Another threat to validity is the possible overhead caused by additional features found in DiSL and AspectJ. DiSL wraps its instrumented code with exception handling and a dynamic bypass check. We mitigate this by disabling any source of additional overhead in DiSL to put it in the best configuration for performance comparison. Whereas, AspectJ generates additional classes that contain the implemented advice at each instrumentation point. However, there is nothing we can do as AspectJ is not customizable and not compatible with inline instrumentation.
Table 4 For each benchmark in the DaCapo experiment, the table reports the number of classes in the scope of instrumentation (Scope), the instrumented classes (Instr.), the original (Ref.) and generated bytecode size and overhead per tool, and the number of emitted events, (#) for BISM and DiSL, and AspectJ separately.

| Benchmark | Scope | Instr. | Ref. | BISM | DiSL | AspectJ | Events (M) |
|-----------|-------|-------|------|------|------|---------|-------------|
| avrora    | 1,550 | 35    | 257  | 264  | 2.72 | 270     | 5.06        |
| batik     | 2,689 | 136   | 1,544| 1,572| 1.81 | 1,588   | 2.85        |
| fop       | 1,336 | 172   | 1,784| 1,808| 1.35 | 1,876   | 5.16        |
| h2        | 472   | 61    | 694  | 704  | 1.44 | 720     | 3.75        |
| pmd       | 721   | 90    | 756  | 774  | 2.38 | 794     | 5.03        |
| sunflow   | 221   | 8     | 69   | 71   | 2.90 | 74      | 7.25        |
| xalan     | 661   | 9     | 100  | 101  | 1.00 | 103     | 3.00        |

9 Related work and discussion

Low-level code instrumentation is widely used for monitoring software and implementing dynamic analysis tools. Several instrumentation tools and frameworks in different programming languages were developed. RV tools [3, 15] generally take instrumentation for granted and rely on some tools for this purpose. In the case of Java programs, AspectJ has been the main tool used.

In the following, we focus our comparison on Java instrumentation tools. Nevertheless, there are several tools to instrument programs in different programming languages. For instance, to instrument C/C++ programs, AspectC/C++ [9, 33] (high-level) and LLVM [26] (low-level) are widely used.

9.1 Instrumentation tools

Table 5 shows a comparison between some of the main tools for instrumenting Java programs among from which DiSL and AspectJ are the closest to BISM. The comparison considers some important features for Java program (bytecode-) instrumentation, including bytecode visibility, the ability to insert bytecode instructions, and whether using the tool requires proficiency in bytecode, level of abstraction, and whether the instrumentation mechanism follows the AOP paradigm.

BCEL [1] enables developers to perform static analysis and dynamically create and modify Java classes at runtime. It is suitable for compilers, profilers, and bytecode optimization tools. Its API consists of a package for analyzing a Java class without having the source code, a package to generate or modify class objects dynamically, and tools to display the target class or convert it into HTML format or assembly language. BCEL does not follow the AOP paradigm and requires proficiency in bytecode.

ASM [7] is a Java bytecode manipulation framework utilized by several tools, including BISM. ASM offers two APIs that can be used interchangeably to parse, load, and modify classes. Its efficient visitor-based API allows the traversal and manipulation of the bytecode. However, to use ASM, a developer must deal with the low-level details of bytecode instructions and the JVM. BISM is implemented on top of the ASM API to provide a superset of its features and offer the user a higher level of abstraction. In particular, for simple instrumentation tasks, the user is not concerned with traversing the bytecode in the correct order, and instead can choose any selector and add advice methods. Moreover, accessing dynamic context and retrieving their values can be a tedious task. It requires analyzing the code and adding bytecode instructions such as stack duplication and local variables assignments at the correct locations. BISM handles these tasks and allows the user to retrieve dynamic context via its dynamic context objects. The same reasoning applies to the advice methods, where the user inserts advice into the base program and BISM handles the generation and weaving of the needed bytecode instructions.

Javassist [8] is a class library for editing bytecode in Java. It provides developers the ability to modify Java classes at runtime when being loaded by the JVM. Javassist provides two levels of API: source level and bytecode level. The source-level API does not require the developer to know Java bytecode.

CGLIB [4] is a code generation library that allows developers to extend Java classes and add new classes at runtime. CGLIB makes use of ASM and some other tools. It provides some level of abstraction and can be used without having profound knowledge about bytecode.

Soot [35] is a framework for analyzing and transforming Java and Android applications. It offers four complementary intermediate code representations, each targeting a set of problems. Soot is mainly used for bytecode optimization static analysis and provides static analyses such as call-graph construction, data-flow analysis, taint analysis, and points-to analysis. When considering a bytecode manipulation library
for BISM, we considered Soot, but finally decided to use ASM due to its fast performance and compact size.

DiSL [27] is a bytecode-level instrumentation framework designed for dynamic program analysis. DiSL adopts an aspect-oriented paradigm. It provides an extensible joinpoint model by providing an extensible library for implementing custom pointcuts (markers). Even though BISM provides a fixed set of pointcuts (selectors), it performs static analysis on target programs to offer out-of-the-box additional and needed control-flow pointcuts with richer static context objects. Both tools do not offer dynamic pointcuts such as cflow, this, args and if from AspectJ. As for dynamic context objects, both BISM and DiSL provide equal access. However, DiSL provides typed dynamic objects. Also, both tools are capable of inserting synthetic local variables. Both BISM and DiSL require basic knowledge about bytecode semantics from their users. In DiSL, writing custom markers and context objects also requires additional ASM syntax knowledge. However, DiSL does not allow the insertion of arbitrary bytecode instructions but provides a mechanism to write custom transformers in ASM that runs before instrumentation. Whereas BISM allows to directly insert bytecode instructions, as seen in Sect. 8.1. Such a mechanism is essential in many runtime verification scenarios. All in all, DiSL provides more features (mainly targeted for writing dynamic analysis tools) and enables dynamic dispatch amongst multiple instrumentations and analysis without interference [5], while BISM is more lightweight, as shown by our evaluation. Moreover, DiSL runs a separate virtual machine for instrumentation, while BISM runs as a standalone tool and requires no installation.

AspectJ [25] is the standard aspect-oriented programming [24] framework, highly adopted for instrumenting Java applications. It provides a high-level language used in several domains like monitoring, debugging, and logging. AspectJ provides a complex joinpoint model with an expressive pointcut expression language and dynamic pointcuts. However, AspectJ cannot capture bytecode instructions and basic blocks joinpoints, making several instrumentation tasks impossible. With BISM, developers can target single bytecode instructions and basic block levels, and they can access richer static joinpoint information. Moreover, BISM provides access to local variables and stack values. Furthermore, AspectJ introduces a significant instrumentation overhead, as seen in Sect. 8.3, and provides less control on where instrumentation snippets get inlined. In BISM, the advice methods are weaved with minimal bytecode instructions and are always inlined next to the targeted regions.

9.2 Transformer composition

Composition and interference problems in aspect-oriented programming have been studied in the literature. Interference between different aspects is commonly addressed as aspect interactions and aspect interference. The main objective is to detect places of interaction between different aspects (collision of transformers in BISM) and provide mechanisms to resolve conflicts. In [11], a framework for the detection and resolution of aspect interactions is presented. The work provides a formal model for aspect weaving and a framework for detecting and resolving conflicts between aspects using static analysis. In [34], the work focuses on unexpected behavior of combined advice (advice interference). They show that controlling the order of execution of advice is not enough in some instances. They propose an AspectJ extension with a new resolver around advice for resolving interference where there is a conflict. The introduced resolver can be implemented separately and composed to resolve interference between other resolvers. BISM provides a built-in feature to capture transformer collision after a run. However, we do not provide a mechanism for resolving conflicts, which can be addressed in our future work.

Composition conflicts are also studied in literature concerning the base program and a single aspect. In [19] and [20], composition conflicts related to introductions to the base program are modeled and detected using a graph-based approach. Introductions are constructs that affect the structure of a class, such as changing the inheritance structure, adding and removing methods. In BISM, such introductions are possible since the user is free to use the ASM structure and modify the class structure. However, we do not address such conflicts and keep the user responsible for avoiding them.

10 Conclusions and future work

10.1 Conclusions

BISM is an effective tool for low-level and control-flow-aware instrumentation. BISM is complementary to DiSL, which is better suited for dynamic analysis (e.g., profiling). We demonstrate the versatility of BISM on several simple use cases. Our first evaluation (Sect. 8.1) let us observe a significant advantage of BISM over DiSL due to BISM’s ability to insert bytecode instructions directly, optimizing the instrumentation. Our second and third evaluations (Sect. 8.2 and Sect. 8.3) confirm that BISM is a lightweight tool that can be used generally and efficiently in runtime verification. We notice a similar bytecode performance between BISM and DiSL after build-time instrumentation, since in both tools the instrumentation (monitor invocation) is always inlined next to the joinpoints. On the other hand, AspectJ advice is located in external classes, and the base program is instrumented to call these external classes at the joinpoints.
Table 5 Comparison of some of the main tools for instrumenting Java programs along with some user-oriented features. A checkmark (✓) indicates that the tool provides the feature. A cross mark (✗) indicates that the tool does not provide the feature. A mixed checkmark/cross mark (✓✗) indicates that the tool partially provides the feature.

| Feature                              | BCEL [1] | ASM [7] | Javassist [8] | CGLIB [4] | Soot [35] | DiSL [27] | AspectJ [25] | BISM [32] |
|--------------------------------------|----------|---------|---------------|-----------|----------|-----------|------------|----------|
| Provides Bytecode Visibility         | ✓        | ✓       | ✓             | ✓         | ✓        | ✓         | ✓          | ✓        |
| Allows Bytecode Insertion            | ✓        | ✓       | ✓             | ✓         | ✓        | x         | ✓          | ✓        |
| Requires no Bytecode Proficiency     | ✓        | x       | ✓             | ✓         | ✓        | ✓         | ✓          | ✓        |
| Provides high-Level Abstraction      | x        | ✓       | ✓             | ✓         | ✓        | ✓         | ✓          | ✓        |
| Follows AOP Paradigm                 | ✓        | x       | x             | ✓         | ✓        | ✓         | ✓          | ✓        |

In load-time instrumentation, the gap between BISM and DiSL is smaller in benchmarks with many classes in scope and a small number of instrumented classes. This stems from the fact that BISM performs a complete analysis of the classes in scope to generate its static context. While DiSL generates static context only after marking the needed regions, which is more efficient.

Overall, we demonstrated that BISM can be used as an alternative to AspectJ and DiSL for lightweight and expressive runtime verification and even runtime enforcement (cf. [12, 13, 16]) due to its bytecode insertion capability, equivalent or better performance, and ease of use. The reported use cases also demonstrate BISM versatility in providing support for static and dynamic analysis tools. In addition to verification and enforcement, BISM provides easy access to program information and powerful modification primitives without requiring the source code.

10.2 Future work

We foresee several research avenues related to BISM, which can be split into two categories.

The first category relates to improvements of BISM itself. We plan on extending the BISM language by adding more features to it, such as selector guards that will facilitate the filtering of joinpoints to the user. Guards can be annotations that decorate selectors. They allow users to specify a filter on important static information such as scope, method signature, opcode for instruction, and others. Also, the language can be expanded to add a declarative domain-specific language for specifying simple instrumentation directives. This will allow users to write certain instrumentation specifications without the need to implement a custom transformer. We will also add more advice methods to instance method invocations.

The second category relates to the use of BISM as a support for static and dynamic analysis. As shown by our performance evaluation, BISM is more efficient than the long-used AspectJ instrumentation framework for runtime verification. It is thus desirable to investigate the performance improvements that runtime verification tools could gain by using BISM as an alternative instrumentation tool. Moreover, since BISM is capable of retrieving some static information about the program, static analysis tools can then be developed as transformers in BISM. Such static analysis would not need the source code and execute using only the bytecode of the target programs. Static analysis can also be beneficial in combination with a runtime verification approach to, e.g., enrich the information provided to a monitor or reduce the performance overhead of monitors. For instance, BISM can be used to implement a control-flow-aware runtime verification tool. Such an approach could (i) use both low-level control-flow events and higher-level events such as method calls and (ii) leverage some reachability analysis on the control flow. Finally, using the bytecode insertion capabilities of BISM, effective runtime enforcement [13, 16] tools can be implemented.

References

1. Apache Commons: BCEL (byte code engineering library). https://commons.apache.org/proper/commons-bcel, accessed: 2020-06-18
2. Bartocci, E., Falcone, Y., Francelanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics. Lecture Notes in Computer Science, vol. 10457, pp. 1–33. Springer, Berlin (2018)
3. Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. Int. J. Softw. Tools Technol. Transf. 21(1), 51–70 (2019). https://gitlab.inria.fr/crv14/benchmarks/
4. Berlin, S., et al.: CGLIB (byte code generation library). https://github.com/cglib/cglib, accessed: 2021-05-21
5. Binder, W., Moret, P., Tanter, É., Ansaloni, D.: Polymorphic bytecode instrumentation. Softw. Pract. Exp. 46(10), 1351–1380 (2016)
6. Blackburn, S.M., Garner, R., Hoffmann, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A.L., Jump, M., Lee, H.B., Moss,
Havinga, W., Nagy, I., Bergmans, L., Aksit, M.: A graph-based tool to implement adaptable systems. In: Adaptable and Extensible Component Systems (2002). https://asm.ow2.io

Bruneton, E., Lenglet, R., Coupaye, T.: ASM: a code manipulation tool to implement adaptable systems. In: Adaptable and Extensible Component Systems (2002). https://asm.ow2.io

Chiba, S.: Load-time structural reflection in Java. In: Bertino, E. (ed.) ECOOP 2000 - Object-Oriented Programming, 14th European Conference, Sophia Antipolis and Cannes, France, June 12-16, 2000. Proceedings. Lecture Notes in Computer Science, vol. 1850, pp. 313–336. Springer, Berlin (2000)

Coady, Y., Kiczales, G.,Feeley, M.J., Smolyn, G.: Using AspectC to improve the modularity of path-specific customization in operating system code. In: Tjoa, A.M., Gruhn, V. (eds.) Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001, pp. 88–98. ACM, New York (2001)

DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. Computer (1978)

Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 2487, pp. 173–188 (2002)

Falcone, Y.: You should better enforce than verify. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, J., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, M., (eds.) Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6418, pp. 89–105. Springer, Berlin (2010)

Falcone, Y., Pinisetty, S.: On the runtime enforcement of timed properties. In: Finkbeiner, B., Mariani, L. (eds.) Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019. Proceedings. Lecture Notes in Computer Science, vol. 11757, pp. 48–69. Springer, Berlin (2019)

Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D.A., Kalus, G. (eds.) Engineering Dependable Software Systems. NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 34, pp. 141–175. IOS Press, Amsterdam (2013)

Falcone, Y., Krstic, S., Reger, G., Tryptel, D.: A taxonomy for classifying runtime verification tools. In: Colombro, C., Leucker, M. (eds.) Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018. Proceedings. Lecture Notes in Computer Science, vol. 11237, pp. 241–262. Springer, Berlin (2018)

Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics. Lecture Notes in Computer Science, vol. 10457, pp. 103–134. Springer, Berlin (2018)

Falcone, Y., Krstic, S., Reger, G., Tryptel, D.: A taxonomy for classifying runtime verification tools. Int. J. Softw. Tools Technol. Transf. 23(2), 255–284 (2021)

Fitzpatrick, J.: Applying the ABC metric to C, C++, and Java. In: C++ Report, pp. 245–264 (012000)

Havinaga, W., Nagy, I., Bergmans, L.M.J.: An analysis of aspect composition problems. Tech. Rep. Technical Report IAI-TR-2006-6, (2006)

Havinaga, W., Nagy, I., Bergmans, L., Aksit, M.: A graph-based approach to modeling and detecting composition conflicts related to introductions. ACM Int. Conf. Proc. Ser. 208, 85–95 (2007)

Honglei, T., Wei, S., Yanan, Z.: The research on software metrics and software complexity metrics. In: 2009 International Forum on Computer Science-Technology and Applications (2009)

Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. (2011)

Khoshgoftaar, T.M., Allen, E.B., Yuan, X., Jones, W.D., Hudepohl, J.P.: Assessing uncertain predictions of software quality. In: Proceedings Sixth International Software Metrics Symposium (Cat. No.PR00403) (1999)

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtjer, J., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP’97. LNCS, vol. 1241, pp. 220–242. Springer, Berlin (1997)

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

Lattner, C., Adve, V.S.: LLVM: a compilation framework for whole-program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), San Jose, CA, USA, 20-24 March 2004, pp. 75–88. IEEE Comput. Soc., Los Alamitos (2004)

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

McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. (1976)

Ofliett, A.J., Untch, R.H.: Mutation 2000: uniting the orthogonal. In: Mutation Testing for the New Century, pp. 34–44. Springer, Boston (2001)

Reger, G., Hallé, S., Falcone, Y.: Third international competition on runtime verification - CRV 2016. In: Falcone, Y., Sánchez, C. (eds.) Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016. Proceedings. Lecture Notes in Computer Science, vol. 10012, pp. 21–37. Springer, Berlin (2016)

Soueidi, C., Kassem, A., Falcone, Y.: BISM: bytecode-level instrumentation for software monitoring. In: Deshmukh, J., Nickovic, D. (eds.) Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020. Proceedings. Lecture Notes in Computer Science, vol. 12399, pp. 323–335. Springer, Berlin (2020)

Soueidi, C., Kassem, A., Falcone, Y.: BISM: Bytecode-Level Instrumentation for Software Monitoring. https://gitlab.inria.fr/bismos/bism-public/

Spinczyk, O., Lohmann, D., Urban, M.: AspectC++: an AOP extension for C. Softw. Dev. J. 01 (2005)

Takeyama, F., Chiba, S.: An advice for advice composition in AspectJ. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 2487, pp. 141–175. IOS Press, Amsterdam (2013)

Vallet-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: a Java bytecode optimization framework. In: CASCON First Decade High Impact Papers. CASCON ’10, pp. 214–224. IBM Press, Raleigh (2010)

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.