Efficient and Expressive Bytecode-Level Instrumentation for Java Programs

Chukri Soueidi, Marius Monnier, Ali Kassem, Yliès Falcone

▶ **To cite this version:**

Chukri Soueidi, Marius Monnier, Ali Kassem, Yliès Falcone. Efficient and Expressive Bytecode-Level Instrumentation for Java Programs. 2022. hal-03533152

HAL Id: hal-03533152
https://inria.hal.science/hal-03533152v1

Preprint submitted on 20 Jan 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

EFFICIENT AND EXPRESSIVE BYTECODE-LEVEL INSTRUMENTATION FOR JAVA PROGRAMS

A PREPRINT

Chukri Soueidi
Univ. Grenoble Alpes, Inria,
CNRS, Grenoble INP, LIG
38000 Grenoble
France
chukri.a.soueidi@inria.fr

Marius Monnier
Univ. Grenoble Alpes, Inria,
CNRS, Grenoble INP, LIG
38000 Grenoble
France
marius.monnier@inria.fr

Ali Kassem
Univ. Grenoble Alpes, Inria,
CNRS, Grenoble INP, LIG
38000 Grenoble
France
ali.kassem@inria.fr

Yliès Falcone
Univ. Grenoble Alpes, Inria,
CNRS, Grenoble INP, LIG
38000 Grenoble
France
ylies.falcone@inria.fr

June 2, 2021

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 language is inspired by the aspect-oriented programming paradigm in modularizing instrumentation into separate *transformers*, that encapsulate joinpoint selection and advice inlining. 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 their collision in 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 BISM instrumentation incurs low runtime and memory overheads.

***K*eywords** 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 workflow [1], 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. Another 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) [2] is a popular and convenient paradigm for instrumenting a program. AOP advocates the modularization of crosscutting concerns such as instrumentation. For Java programs, runtime verification tools [3, 4] have long relied on AspectJ [5], which is one of the reference 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. 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 Section 8.1, we demonstrate an instrumentation scenario that cannot be achieved with AspectJ. Moreover, 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 to define complex compile-time computations.

Nevertheless, there are several low-level bytecode manipulation frameworks such as ASM [6] (in reference to the __asm__ C++ operator) and BCEL [7] (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. Other bytecode instrumentation frameworks, from which DiSL [8] is the most remarkable, enable flexible low-level instrumentation and, at the same time, provide 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 before 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 Section 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. The language is inspired by AOP but adopts an instrumentation model that is more directed towards 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. BISM 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, BISM provides several control-flow properties, such as capturing conditional jump branches and retrieving successor and predecessor basic blocks. Such features can provide support to tools relying on a control-flow analysis, 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 of 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 BISM can be used to instrument a program to detect test inversion attacks. For this purpose, we use BISM to instrument AES (Advanced Encryption Standard). The second and third experiments demonstrate using BISM to instrument for general runtime verification cases. In the second experiment, we instrument a simplified financial transaction system to check for various properties from [4]. In the third experiment, we instrument seven applications from the DaCapo benchmark [9] 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.

This paper extends a paper that appeared in the proceedings of the 20$^{\text{th}}$ Runtime Verification conference [10]. The paper provides the following additional contributions:

- new language features: meta selectors, support for synthetic local arrays, and config files (in Section 3);
- instrumentation model, underlying shadows and equivalence (in Section 4 and Section 4.4).
- transformer composition, collision detection and visibility control of advice (in Section 5);
- use cases to demonstrate the usage of BISM in different contexts (in Section 6);
- an additional experiment demonstrating BISM effectiveness using a benchmark from the competitions on runtime verification [4, 11] (in Section 8.2).

Paper organization. Section 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 Design Goals and Features

BISM is a bytecode instrumentation tool for Java programs implemented on top of ASM [6]. It provides an instrumentation language that is expressive like ASM and provides the user with a high level of abstraction as inspired by AOP. 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 Section 4). In particular, BISM does not have the whole notion of pointcuts that allows specifying a predicate to match different joinpoints. Instead, BISM 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 Java programs.

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 allows bytecode insertion, method invocation, and printing.

Access to program context. BISM offers rich access to complete static information about instructions, basic blocks, methods, and classes. It also offers dynamic context objects that provide access to values that will only be available at runtime, such as local variables, stack values, method arguments, and results. Moreover, BISM allows accessing instance 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 for inserting raw bytecode instructions by using the ASM data types. In this case, it is the responsibility of the user to write instrumentation 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 cost of possibly 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];
- easily performs dynamic program analysis (e.g., profiling, debugging).

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). It is generally faster to execute the instrumented program than to load BISM as an agent.

Portability and ease of use. BISM is a lightweight tool designed in Java and fitting in a single jar application 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 include the BISM jar file path to the *classpath* (Java runtime environment variables) to compile new custom transformers. BISM has been successfully tested on various operating systems and even embedded devices. As it has no installation requirements, we believe it is portable and usable in any environment.

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* (Section 3.1) to select joinpoints of interest, static and dynamic context objects (Section 3.2 and Section 3.3) which retrieve relevant information from these points, and *advice* methods (Section 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 execution before a bytecode instruction.
- `AfterInstruction` captures execution after a bytecode instruction. If the instruction is the exit point of a basic block, it captures the code before the instruction.
- `BeforeMethodCall` captures execution just before a method call instruction and after loading all needed values on the stack.
- `AfterMethodCall` captures execution immediately after a method call instruction and before storing the return value from the stack, if any.

Method. BISM also provides two method-related selectors:

- `OnMethodEnter` captures execution on method entry block, same execution rules as OnBasicBlockEnter.
- `OnMethodExit` captures 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 BISM and of instrumentation. To the best of our knowledge, there is no exhaustive list of classes with the before-mentioned flag.

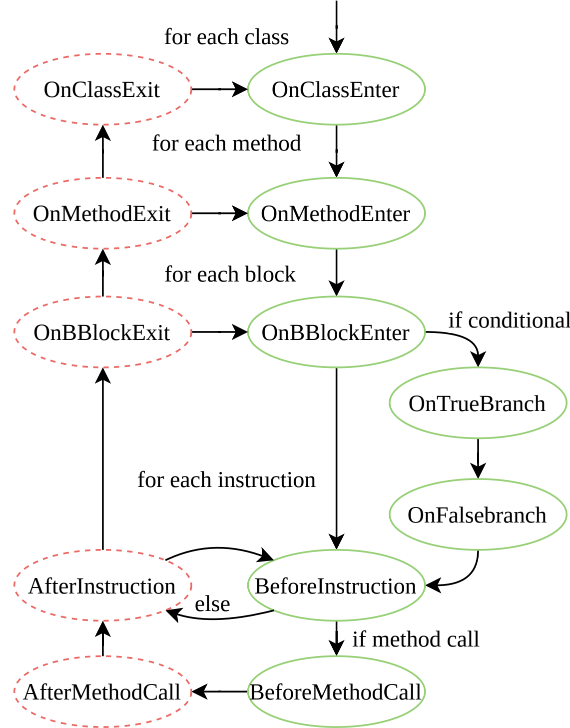


Figure 1: Instrumentation Loop of BISM.

Basic block. In addition to the previous selectors, BISM provides basic block-related selectors which ease capturing control-flow related execution points:

- `OnBasicBlockEnter` captures execution when entering the block, before the first real instruction[4].
- `OnBasicBlockExit` captures execution after the last instruction of a basic block; except when last instruction is a JUMP/RETURN/THROW instruction, then it captures the code before the instruction.
- `OnTrueBranchEnter` captures execution on the entry of a successor block after a conditional jump on True evaluation.
- `OnFalseBranchEnter` captures execution on the entry of a successor block after a conditional jump on False evaluation.

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 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 Figure 1. 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 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

[4]Real instructions are instructions that actually get executed, as opposed to some special Java bytecode instructions such as labels or line number instructions.

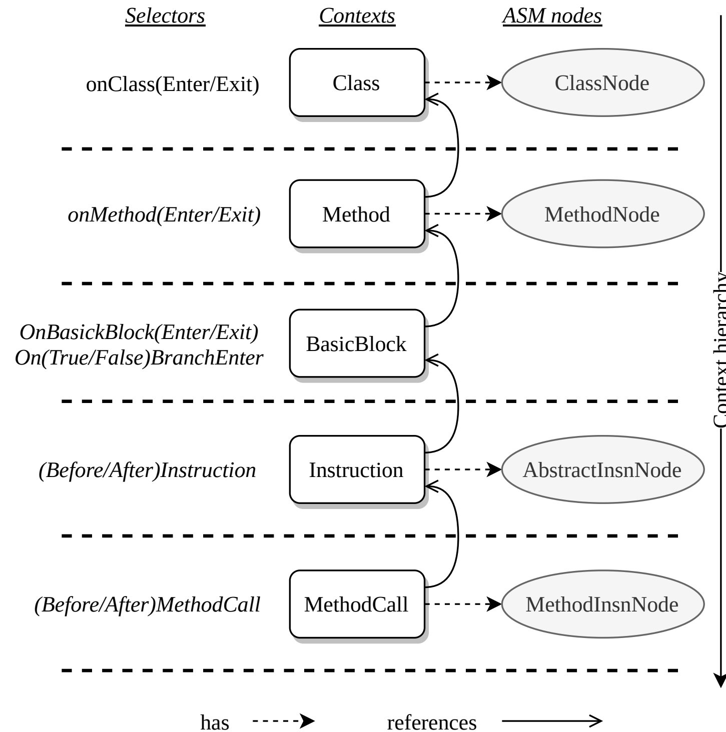


Figure 2: The static context tree related to selectors and ASM nodes.

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 Figure 2. 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.

```
public void beforeMethodCall(MethodCall mc, ...){
   Instruction i = mc.ins; //Access the surrounding Instruction static context
   BasicBlock b = i.basicBlock; //Access the surrounding BasicBlock context
   Method m = b.method; //Access the surrounding Method context
   ClassContext c = m.classContext; //Access the surrounding Class context
}
```

Listing 1: Static context objects hierarchy.

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 of 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.

MethodCall context. This 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.

BasicBlock 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 instructions of this block which are executable ones (not labels);

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 an annotation on the method;
- 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.

```
public 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.

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 [12]. 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;
- `getThreadName()`: 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;
- `getStatic/InstanceField(String)`: returns a reference to an instance/static field in the class being instrumented.

BISM gives access to method-relative information. The runtime arguments passed to a method can be retrieved using `getMethodArgs(int)`. The method result (return value) can be retrieved using `getMethodResult()`. The object on which the method is called can be retrieved using `getMethodReceiver()`.

It is also possible to add new local variables of primitive types with the call `addLocalVariable(Object value)` The scope of the added variables is the method where they are created. This is useful for different purposes like to pass data across selectors. Local arrays could also be added with the `createLocalArray(Method, Class)` method. BISM weaves the necessary bytecode and returns a dynamic value to query and update freely, such as clearing and appending elements. It is particularly useful when there is a need to pass objects between selectors in a method without knowing how much space will be needed at runtime.

Listing 3 presents a transformer the uses the selector `afterMethodCall` to capture the return of an `Iterator` created from a `List` object. It uses the dynamic context object using `MethodCallDynamicContext dc` provided to the selector to retrieve the dynamic data. The example also shows how to limit the scope to a specific method using an if-statement on the static context.

```
@Override
public void afterMethodCall(MethodCall mc, MethodCallDynamicContext dc){
  if (mc.methodName.equals("iterator") && mc.methodOwner.endsWith("List")) {
     //Access to dynamic data
     DynamicValue callingClass = dc.getThis(mc);
     DynamicValue list = dc.getMethodTarget(mc);
     DynamicValue iterator = dc.getMethodResult(mc);

     //Invoking a monitor
     StaticInvocation sti = new StaticInvocation("IteratorMonitor", "iteratorCreation");
     sti.addParameter(callingClass);
     sti.addParameter(list);
     sti.addParameter(iterator);
     invoke(sti);
  }
}
```

Listing 3: A transformer that intercepts the creation of an iterator from a `List`.

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 and (ii)

`insert` methods for inserting bytecode instructions. These methods are compiled by BISM into bytecode instructions and inlined 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 instrument the base program to call an external static method (here named `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.

Raw bytecode instructions. Inserting raw bytecode instructions can be achieved with the `insert` methods. When used, the developer's responsibility is 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 *(scope=java.util.List.*, java.util.Iterator.next)* will instrument all methods in the `List` class and only the next method in the `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 in the program given by *shadows*. We then introduce the equivalence between shadow, which is used when composing transformers (Section 5).

4.1 Overview

A *joinpoint* 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 is only interested in some of the joinpoints generated by the program execution. At these joinpoints, the user-specified advice is executed.

When instrumenting a program, we inject extra instructions at specific bytecode locations in the code of the base program. Bytecode regions denote the locations where instrumentation can happen. The user implements selectors (Section 3.1) to select joinpoints using lexical conditions that mark 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 (*before* and *after*), or basic blocks and methods along with the *enter* or *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 where instrumentation can happen. BISM internally operates on shadows to extract static information and delimit the regions where advice will be woven. Shadows are pairs consisting of bytecode instructions along with a specified direction (*before* and *after*), or basic blocks and methods along with an *enter* or *exit* direction.

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

$$\begin{aligned} Shadows = \quad & (\{\text{before}, \text{after}\} \times Instrs) \\ \cup \; & (\{\text{enter}, \text{exit}\} \times Blocks) \\ \cup \; & (\{\text{enter}, \text{exit}\} \times Methods) \end{aligned}$$

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

Definition 4.1 (Method Shadows)**.** $Shadows_m$ denotes the set of all shadows in a method.

$$\begin{aligned} Shadows_m = \quad & (\{\text{before}, \text{after}\} \times m.\text{instrs}) \\ \cup \; & (\{\text{enter}, \text{exit}\} \times m.\text{blocks}) \\ \cup \; & \{\langle \text{enter}, m \rangle, \langle \text{exit}, m \rangle\} \end{aligned}$$

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.

```
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 4: A method calling an Iterator.

Example 4.1 (Method shadows)**.** Listing 4 contains a Java method `m` that creates a `List l` with an associated `Iterator i`. The method checks if `i.hasNext()` and calls `i.next()`. Listing 5 shows the (simplified) bytecode for method m in black font. We also show the shadows identified by BISM, added in the colors blue, red, and olive green. There are two shadows for the method entry, and exit points showed. Two shadows for each basic block (the if-statement results in having three basic blocks), and for each instruction, two shadows to delimit the region before it and after it (we omitted most of the instruction shadows for brevity).

4.3 Selectors Matching Shadows

Each BISM selector matches a specific subset of the shadows. Selectors `OnMethodEnter` and `OnMethodExit` respectively match the shadows $\langle \text{enter}, m \rangle$ and $\langle \text{exit}, m \rangle$ for each method $m \in Methods$. Selectors `OnBasicBlockEnter` and `OnBasicBlockExit` respectively match the shadows $\langle \text{enter}, b \rangle$ and $\langle \text{exit}, b \rangle$ for

```
m() {
⟨enter, m0⟩
⟨enter, b0⟩
⟨before, i0⟩
_new ArrayList
⟨after, i0⟩
⟨before, i1⟩
dup
invokespecial ArrayList.init ()V
astore 1
aload 1
ldc A
invoke List.add (Object;)Z
pop
aload 1
⟨before, i9⟩
invokeinterface List.iterator ()Iterator;
⟨after, i9⟩
astore 2
aload 2
invokeinterface Iterator.hasNext ()Z
⟨exit, b0⟩
ifeq L0
⟨enter, b1⟩
aload 2
invokeinterface Iterator.next ()Object;
pop
⟨exit, b1⟩
L0
⟨enter, b2⟩
getstatic System.out, PrintStream;
ldc done
invokevirtual PrintStream.print (String;)V
⟨exit, b2⟩
⟨exit, m0⟩
return
}
```

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

each basic block $b \in \mathit{Blocks}$. Selectors `OnTrueBranch` and `OnFalseBranch` match special instances of the shadows $\langle \text{enter}, b \rangle$ where basic block b has a predecessor block ending with a conditional jump. Selectors `BeforeInstruction` and `AfterInstruction` respectively match the shadows $\langle \text{before}, i \rangle$ and $\langle \text{after}, i \rangle$ for each instruction $i \in \mathit{Instrs}$. Selectors `BeforeMethodCall` and `AfterMethodCall` respectively match the shadows $\langle \text{before}, i \rangle$ and $\langle \text{after}, i \rangle$ for each instruction $i \in \mathit{Instrs}$ that is a method invocation instruction. We demonstrate the matching in the following example.

Example 4.2 (Selectors matching shadows)**.** In this example, we show the shadows matched by BISM selectors when applying different transformers to the Java program from Example 4.1. 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 Figure 1. This allows the user to implement compile-time static analyzers, as we will see in Section 6. A transformer can be implemented to analyze the code without instrumenting it. Selectors play the role of Visitor methods where the user can write custom code for the static analyzer without even instrumenting the base program using the advice methods(Section 3.4). This enables combining static analysis with runtime verification using transformer composition (Section 5) and helps in optimizing instrumentation.

4.4 Equivalence Between Shadows

Since a bytecode region can be referred to by multiple shadows, to detect when two transformers contain selectors with shadows that mark the same regions, we define the notion of equivalence between shadows. The user may unintentionally target the same regions using different selectors in a transformer. The choice of selector used depends on the static/dynamic context needed by the user; since each selector exposes different static/dynamic context objects. We define the equivalence relation over shadows below and give an illustrative example.

Definition 4.2 (Equivalence Relation over Shadows)**.** The equivalence relation over shadows in a method m is defined as follows:

$$\equiv_{\mathrm{s}}^{m} \subseteq \mathit{Shadows}_m \times \mathit{Shadows}_m$$

$$\begin{aligned}
&\stackrel{\text{def}}{=} \{\langle \text{enter}, m\rangle, \langle \text{enter}, m.\text{entryBlock}\rangle\ \} && (1)\\
&\quad \cup \{\ \langle \text{exit}, b\rangle, \langle \text{exit}, m\rangle \mid b \in m.\text{exitBlocks}\ \} && (2)\\
&\quad \cup \{\ \langle \text{enter}, b\rangle, \langle \text{before}, i\rangle \mid b \in m.\text{blocks}\ \wedge\ i.\text{index} = b.\text{first.index}\} && (3)\\
&\quad \cup \{\ \langle \text{after}, i\rangle, \langle \text{exit}, b\rangle \mid b \in m.\text{blocks}\ \wedge\ i.\text{index} = b.\text{last.index}\} && (4)\\
&\quad \cup \{\ \langle \text{after}, i\rangle, \langle \text{before}, i'\rangle \mid \exists k \in [1, \text{size}(m.\text{instrs})] : i.\text{index} = k\ \wedge\ i'.\text{index} = k+1\} && (5)
\end{aligned}$$

In Definition 4.2, line (1) states that the region on method enter is equivalent to the region on basic block enter if the block is the entry block of the method. Line (2) states that the region at a method exit is equivalent to the region at the exits of all basic blocks that are exit blocks in the method. Line (3) states that the region at block entry is equivalent to the region before the first instruction in the block defined by $b.\text{first}$. Line (4) states that the region at the exit of a block is equivalent to the region after the last instruction of the block defined by $b.\text{last}$. Line (5) states that the region after an instruction and before its consecutive are equivalent.

Example 4.3 (Equivalent shadows in a method)**.** Figure 3, 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 4.2. From (1), we have $\langle \text{enter}, m\rangle \equiv_{\mathrm{s}}^{m} \langle \text{enter}, b_1\rangle$. From (2), we have $\langle \text{exit}, b_4\rangle \equiv_{\mathrm{s}}^{m} \langle \text{exit}, b_2\rangle \equiv_{\mathrm{s}}^{m} \langle \text{exit}, m\rangle$. From (3), we have $\langle \text{enter}, b_3\rangle \equiv_{\mathrm{s}}^{m} \langle \text{before}, k\rangle$. From (4), we have $\langle \text{after}, l\rangle \equiv_{\mathrm{s}}^{m} \langle \text{exit}, b\rangle$. From (5), we have $\langle \text{after}, i\rangle \equiv_{\mathrm{s}}^{m} \langle \text{before}, j\rangle$.

5 Transformer Composition

BISM allows users 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 (Section 5.1) and address some concerns that may arise when multiple transformers target the exact program bytecode regions (Section 5.2) and how BISM can help (Section 5.3 and Section 5.4).

5.1 Motivations for Composition

Composition is needed in some cases and optional in others. Transformer composition is obligatory when it is impossible to merge the code of two transformers in one transformer. This situation arises when there is a dependency between transformers, and more than one pass is required to instrument the program. In other cases, we may want to implement separate transformers based on their functionality for a cleaner code. We discuss the two cases.

More than one pass. In many cases, transformations might require multiple passes on the same class. Let us assume 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. BISM can also 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 and applied before the transformer responsible for instrumenting the code for monitoring.

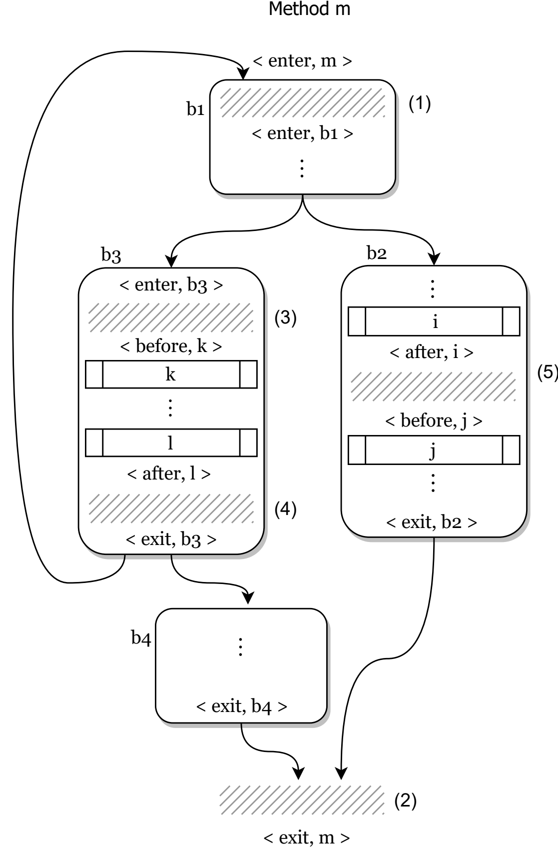


Figure 3: Illustration of shadows and their equivalence relation.

Modularity of transformers. At the core of aspect-oriented programming is increasing modularity by separating concerns. 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 *HasNext*, *UnSafeIterator* and *SafeSyncMap* (see Section 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 determines collision detection after weaving the advice of multiple transformers into the base program. Recall Definition4.1 of the shadows of a method. Let $Shadows_m^t$ represent all the shadows used by BISM to insert the advice for transformer t, in a method m, we have:

$$Shadows_m^t \subseteq Shadows_m$$

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

Definition 5.1 (Transformer Collision)**.** Transformer t collides with transformer t' in method m, iff

$$\exists\, s \in Shadows_m^t \;,\; \exists\, s' \in Shadows_m^{t'} \;:\; s \equiv_{\mathrm{s}}^{m} s'$$

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.

Several concerns may arise from collisions, such as determining the order of execution and the visibility among aspects. In the following sections, we discuss these problems.

5.3 Order Matters

BISM executes the transformations in the order specified by the user. One issue that may arise is that applying the transformations in a different order may exhibit different behavior in the final instrumented program. Let us look at the following example.

Example 5.1 (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 will be 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.

```
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).

In other cases, the execution order is not important to us, even if there is a collision between two transformers. In Section 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 (Section 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 use. In many cases, we may want to hide these 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. A prevalent scenario is to avoid instrumenting previously added code by a different transformer. Let us look at the following example.

Example 5.2 (Hidden transformers)**.** In Listing 7, demonstrates two 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 will be 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 on the transformers hides the newly added instructions from other transformers in a composition.

```
@Hidden
public class CountMethodCalls extends Transformer {
   public void onMethodCall(..) {
      //Invoke methodCounterIncrement
   }
}
@Hidden
public class LogMethodCalls extends Transformer {
   public void onMethodCall(..) {
      //Invoke logger
   }
}
```

Listing 7: `@Hidden` attribute on transformers.

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 implementing 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 (Section 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 mutation, code analysis, runtime verification, dynamic profiling, and logging of programs.

6.1 Mutation of Programs

We consider software testing and, more particularly, mutation testing (see [13] 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. There are various types of mutations of various complexity levels [14, 15]. 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.

6.1.1 Return Mutator: *Value Mutation*

The mutator in Listing 8 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 `m` 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.

6.1.2 Instruction Mutator: *Operator Mutation*

The mutator in Listing 9 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-

```
public void onMethodExit(Method m,...){
//Detecting return type
   if (Type.getReturnType(m.methodNode.desc) == Type.VOID_TYPE)
      return;
   //Remove return value from stack
   if (Type.getReturnType(m.methodNode.desc) .getSize() == 1)
      insert(new InsnNode(Opcodes.POP));
   else
      insert(new InsnNode(Opcodes.POP2));
   //Push fixed return value (0)
   switch(Type.getReturnType(m.methodNode.desc) .getSort()){
      case Type.INT:
         insert(new InsnNode(Opcodes.ICONST_0));
      break;
      ...
   }
}
```

Listing 8: Return mutator.

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.

```
public void 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 9: Generic instruction mutator.

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

- The mutator in Listing 10 targets *conditional operators*, which are detected as *conditional jump* instructions. Another comparison operator replaces conditional operators without changing their destination.
- The mutator in Listing 11 targets binary arithmetic operators on integers. Arithmetic operators are replaced either by a random operator or the complementary one ($-$ and $+$, $\&$ and $|$ for bitstring operators…).

6.1.3 Void Call Mutator: *Statement Mutation*

The mutator in Listing 12 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 `INVOKEX` opcode. To check for return types and unload the parameters differently regarding their sizes, the transformer iterates through the method descriptor[5] available through the static context attribute `mc.methodnode.desc`.

[5]The descriptor is a string representing a type, for a method it permits to access the return and argument types.

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

//Choose a random if which is compatible in term of type and arg count
AbstractInsnNode random(Instruction insIf) {
   if (insIf.opcode >= Opcodes.IFEQ && insIf.opcode <= Opcodes.IFLE)
      return new JumpInsnNode(Opcodes.IFEQ + r.nextInt(Opcodes.IFLE - Opcodes.IFEQ+1),
          ((JumpInsnNode) insIf.node).label);
 ...
}

//Negate the opcode of a given if
AbstractInsnNode negate(Instruction ins){
   if (ins.opcode == Opcodes.IFNULL || ins.opcode % 2 == 1)
      return new JumpInsnNode(ins.opcode +1, ((JumpInsnNode) ins.node).label);
   else
      return new JumpInsnNode(ins.opcode -1, ((JumpInsnNode) ins.node).label);
}
```

Listing 10: Decision mutator.

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

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

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

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

Listing 11: Arithmetic mutator.

6.2 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, upgradable software, and secure coding. In [16, 17], 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, commentaries, 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 access and compute many valuable properties that can be used to compute standard metrics relying on the CFG of methods, the number of variables and 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: Mc Cabe Cyclomatic complexity, ABC Metric, and the count of unused variables.

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

   remove(mc.ins);
}
```

Listing 12: Void call mutator.

Mc Cabe complexity. The Mc Cabe Cyclomatic complexity [18] 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 13, the transformer uses the computed CFG to count the number of conditional edges inside it.

```
int edgeNumber;
public void onMethodEnter(...){
   edgeNumber = 0;
}
public void onBasicBlockExit(BasicBlock bb,...){
   switch (bb.blockType){
      case CONDJUMP:
      case SWITCH:
       edgeNumber++;
   }
}
public void onMethodExit(Method m,...) {
   int c = m.getNumberOfBasicBlocks() - edgeNumber + 2;
   Log("Cyclomatic complexity of "+m.name+" is : "+ c);
}
```

Listing 13: Mc Cabe cyclomatic complexity.

ABC Complexity. To compute the ABC complexity, we only need to classify instructions and basic blocks. Computing the ABC complexity [19] relies on the capability to distinguish between branching, assignments, and conditional jumps. The transformer does it in Listing 14 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 15 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.

6.3 Good Java Practices

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

6.3.1 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. This property Listing 16 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 Section 3.3) retrieves the iterator instance by loading it from the stack into a

```
public void onMethodEnter(Method m,...) {
   A=B=C=0;
   C = m.methodNode.tryCatchBlocks.size(); //To count the try and catch in conditional
}

public void beforeInstruction(Instruction ins,...) {
   if (isAssignInstr(ins))
      A++;
   //Handle branches
   if (ins.opcode == Opcodes.GOTO || ins.opcode == Opcodes.NEW ||
       ins.isBranchingInstruction())
      B++;
}

public void beforeMethodCall(MethodCall mc,...) {
   B++;
}

public void onBasicBlockExit(BasicBlock bb,...) {
   if (bb.blockType == BlockType.CONDJUMP)
      C++;
}

public void onMethodExit(Method m,...) {
   Log("ABC of "+m.name+" is "+ Math.sqrt(A*A+B*B+C*C));
}
```

Listing 14: ABC complexity.

```
public void onMethodEnter(Method m,...) {
   unusedVars = new boolean[ m.methodNode.localVariables.size()];
   Arrays.fill(unusedVars, true);
}

public void beforeInstruction(Instruction ins,...) {
   if (ins.opcode >= Opcodes.LDC && ins.opcode <= Opcodes.SALOAD){
      //Loading a local variable, therefore variable is used
      if (ins.node instanceof VarInsnNode)
         unusedVars[((VarInsnNode) ins.node).var] = false;
      else if (ins.node instanceof IincInsnNode)
         unusedVars[((IincInsnNode) ins.node).var] = false;
   }
}
```

Listing 15: Unused variables.

local variable returning a reference to it in a `DynamicValue` object. Assuming 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`.

6.3.2 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 17, 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.

```
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 16: HasNext instrumentation.

6.4 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 (Section 6.4.1), number of allocated objects (Section 6.4.2), and return types (Section 6.4.3). We do not focus on implementing the profiler tool but only on how to extract context using BISM.

6.4.1 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 calling and called methods. Listing 18 shows the code of a transformer that instruments to extract the *caller* and *callee* classes and method names along with their runtime arguments, at 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`, the `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.

6.4.2 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 19 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.

6.4.3 Return Types

Listing 20 shows how instrument to extract return types from methods. We use the `afterMethodCall` joinpoint and filter using the static context provided `mc.returns` which returns 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

```
@Override
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);

   }
}
@Override
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);
   }
}
```

Listing 17: SafeLock instrumentation.

invocation to the profiler is instrumented, which passes the needed information. We choose to box the return value for a more generic implementation.

6.5 Logging

Logging is a classic example of a cross-cutting concern that is better implemented following the aspect-oriented paradigm. Indeed, by using Java annotations, one can mark methods that require logging and avoid polluting the source code with multiple logging instructions. This way, one can instrument the program and insert the logging instructions only on annotated methods.

Listing 21 shows a transformer that instruments the program to log the execution of selected methods in a program on method entries and exits. One way to mark methods that need to be logged in an application by a developer is by creating a custom annotation and annotating the needed methods. The transformer looks for a hypothetical `@Log` annotation inserted at methods in the base program. The annotation indicates that the method needs to be logged. BISM provides access to annotations on methods through the static context. The `isAnnotated(Log)` returns a boolean flag that indicates if the method is annotated with `@Log`. Then, a simple log message is printed on the console. This example can be extended to extract and log the arguments passed to the method (see Section 6.4.1).

```
LocalArray callerArgs;
LocalArray calleeArgs;

public void onMethodEnter(Method m, MethodDynamicContext dc){
   //Initialize the local arrays
   callerArgs = dc.createLocalArray(m);
   calleeArgs = dc.createLocalArray(m);

   int args = m.getNumberOfArguments();
   DynamicValue dv;
   for (int i = 1; i < args + 1; i++) {
      dv = dc.getMethodArgs(m, i);
      dc.addToLocalArray(callerArgs, dv);
   }
}

public void beforeMethodCall(MethodCall mc, MethodCallDynamicContext dc){
   dc.clearLocalArray(mc, calleeArgs);

   int args = mc.getNumberOfArgs();
   DynamicValue dv;
   for (int i = 1; i < args + 1; i++) {
      dv = dc.getMethodArgs(mc, i);
      dc.addToLocalArray(calleeArgs, dv);
   }

   //Invoke profiler
   StaticInvocation sti = new StaticInvocation("Profiler", "callGraph");
   sti.addParameter(dc.getThreadName(mc));
   sti.addParameter(mc.method.fullName);
   sti.addParameter(callerArgs);
   sti.addParameter(mc.fullName);
   sti.addParameter(calleeArgs);
   invoke(sti);
}
```

Listing 18: Profiling the call graph.

7 BISM Implementation

In this section, we provide some details about BISM implementation. BISM [12] is implemented in Java using about 7,000 LOC and 55 classes distributed in separate modules. It uses ASM for bytecode parsing, analysis, and weaving. BISM can run in two modes: a build-time mode where BISM runs as a standalone application to statically instrument a program, and a load-time mode where BISM is attached to a program as a Java agent. Fig. 4 shows BISM internal workflow.

(1) User Input. In build-time mode, BISM takes a base program bytecode (*.class* or *.jar*) to be instrumented and a list of transformers that specifies the instrumentation logic. In load-time mode, BISM only takes the transformers and instruments all classes being loaded by the JVM. 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.

```
@Override
public void beforeInstruction(Instruction ins,...) {
   //Object creation opcodes
   if (ins.opcode == Opcodes.NEW
   || ins.opcode == Opcodes.NEWARRAY
   || ins.opcode == Opcodes.ANEWARRAY
   || 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 19: Profiling object allocation.

```
 @Override
public void afterMethodCall(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);
   }
}
```

Listing 20: Profiling return types.

Also, if the transformer uses joinpoint `onMethodExit`, all the exit blocks (which terminate with a return opcode) are merged into a single 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 class 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 Figure 1. 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 ASM instructions, BISM emits a weaving error. 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.

```
@Override
public void onMethodEnter(Method m, MethodDynamicContext dc){
   if (m.isAnnotated("Log")) //Checks annotation on method
      println("Entering method: " + m.name);
}

@Override
public void onMethodExit(Method m, MethodDynamicContext dc){
   if (m.isAnnotated("Log"))
      println("Exiting method: " + m.name);
}
```

Listing 21: Logging.

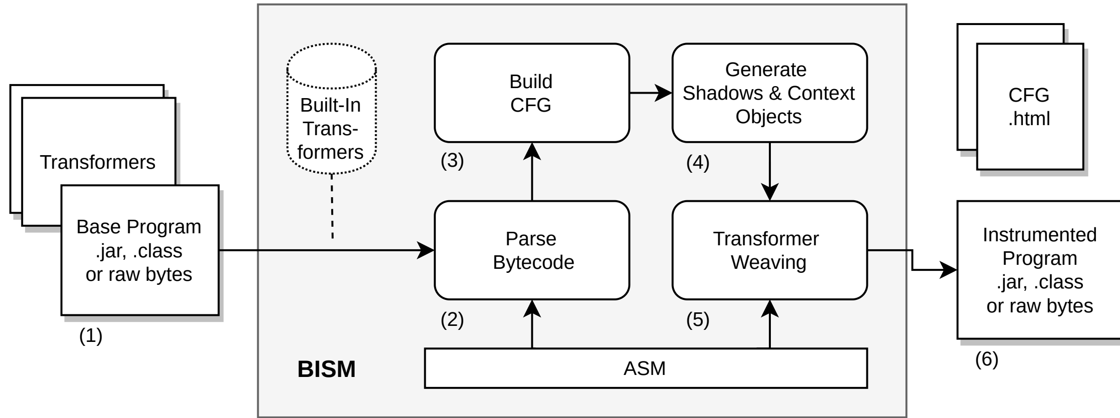


Figure 4: Instrumentation process in BISM.

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[6]. Table 1 illustrates 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 [9]. 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 classically used when evaluating runtime verification tools as it produces events at a high rate.

For the first experiment, the 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 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.

[6]We use the latest versions of DiSL from https://gitlab.ow2.org/disl/disl and AspectJ Weaver 1.9.4.

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].

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	Bytecode	Comparison with	
		Runtime	Used Memory	Bytecode Size	Level	Insertion	AspectJ	DiSL
AES	LT	✓	✓ (-DiSL)	NA	Low (CFG-Level)	✓	NA	✓
	BT		✓	✓				
Transactions	LT	✓	✓ (-DiSL)	NA	High (Method Calls)	✗	✓	✓
	BT		✓	✓				
DaCapo	LT	✓	✓ (-DiSL)	NA	High (Method Calls)	✗	✓	✓
	BT		✓	✓				

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.

In BISM, we use built-in features to duplicate all conditional jumps utilizing the ability to insert raw bytecode instructions. In particular, we use the instrumentation locator `beforeInstruction` to capture conditional jumps. To extract the opcode for each conditional jump, we use the static context object `Instruction`, and to duplicate the operand values on the stack, we use the advice method `insert`[8]. We then use the control-flow instrumentation locators[9] to capture the successor blocks executing after every conditional jump. Finally, at the beginning of these blocks, we utilize `insert` to duplicate the conditional jump instruction.

In DiSL, we implement a custom `StaticContext` object to retrieve information from conditional jump instructions, such as the indices of jump targets and instruction opcodes. Note, we use multiple `BytecodeMarker` snippets to capture all conditional jumps. To retrieve stack values, we use the dynamic context object. We then store the extracted information in synthetic local variables, and we add a flag to specify that a jump has occurred. Finally, on successor blocks, we map opcodes to Java syntax to re-evaluate conditional jumps using switch statements.

[7]More details about the experiments and the material needed to reproduce them can be found at https://gitlab.inria.fr/bism/bism-experiments.

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

[9]`OnTrueBranchEnter,onFalseBranchEnter`.

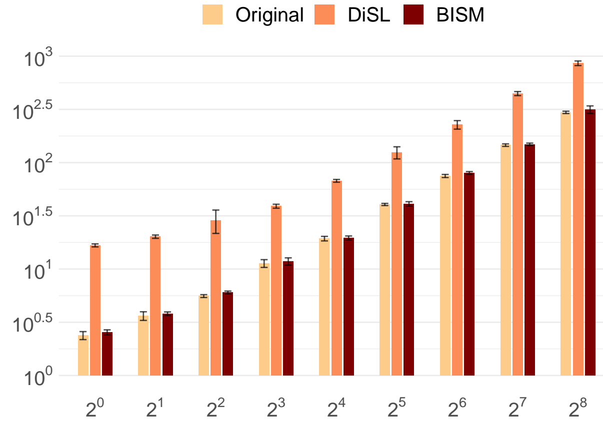
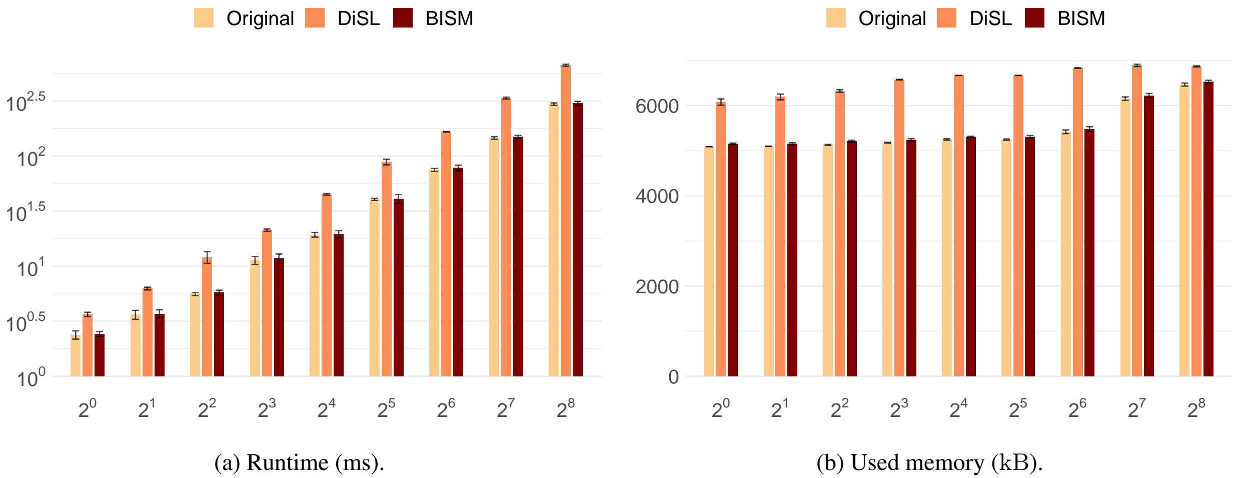


Figure 5: AES load-time instrumentation runtime (ms).



(a) Runtime (ms).

(b) Used memory (kB).

Figure 6: AES build-time instrumentation.

Load-time evaluation. We consider different sizes of the plain text to be encrypted by AES. Figure 5 reports runtime with respect to plain-text size, in load-time mode. BISM shows better performance over DiSL for all plain-text sizes. We do not measure the used memory because DiSL performs instrumentation on a separate JVM process which imposes a huge memory overhead. Also, AspectJ is excluded from this experiment as it cannot capture control-flow events.

Build-time evaluation. We replace the original classes of AES with statically instrumented classes from each tool. Figure 6 reports the runtime and used memory for plain-text size in build-time mode. 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 $9\,\mathrm{kB}$. After instrumentation, the bytecode size is $10\,\mathrm{kB}$ (+11.11%) for BISM, and $128\,\mathrm{kB}$ (+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

Table 2: Number of emitted events in AES experiment.

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

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 [4] 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).

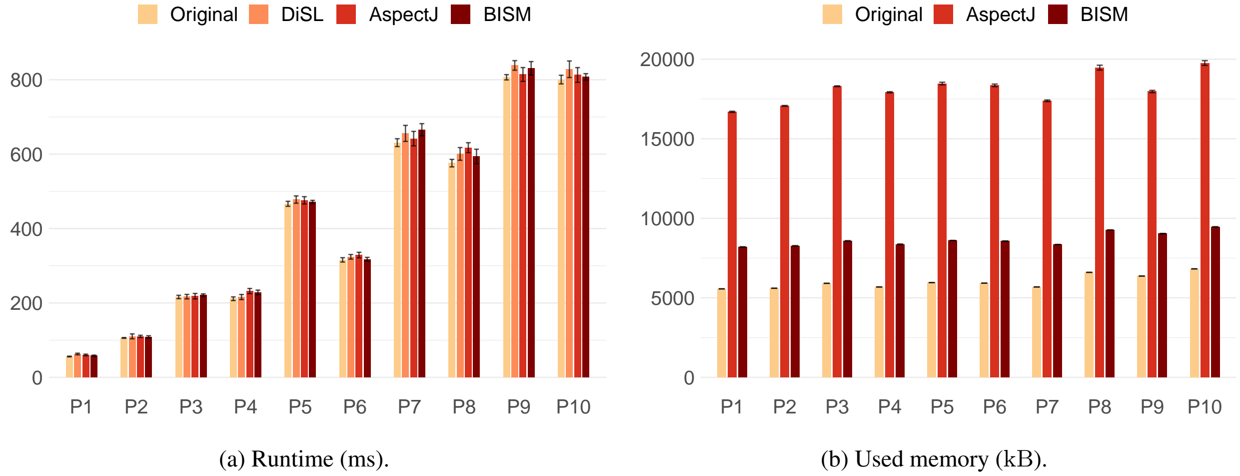
For each property using a set of events, we instrument the financial transaction system to generate those events. Such events mainly correspond on the system to method call with parameters or class field updates. For example, monitoring Property P6 requires the following events: `greylistUser(id)`, `depositFromExternal(id)` and `whitelistUser(id)`, where `id` is a unique user identifier. We implement a set of related scenarios 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 7 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 is due to the fact that each property 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 8 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

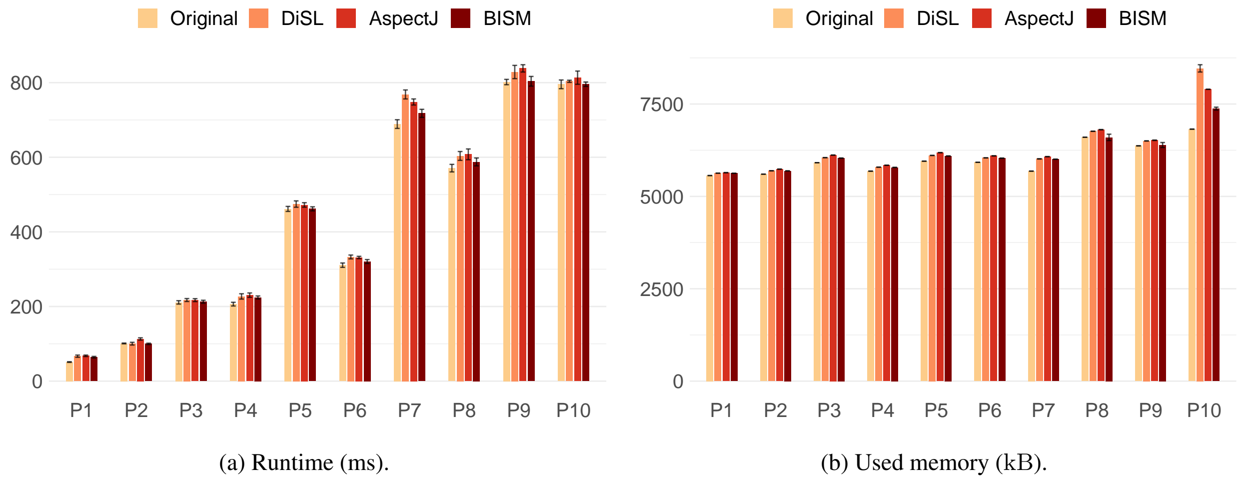
[10] `beforeMethodCall`, `afterMethodCall`.



(a) Runtime (ms).

(b) Used memory (kB).

Figure 7: Financial transaction system load-time instrumentation.



(a) Runtime (ms).

(b) Used memory (kB).

Figure 8: Financial transaction system build-time instrumentation.

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 [9] (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

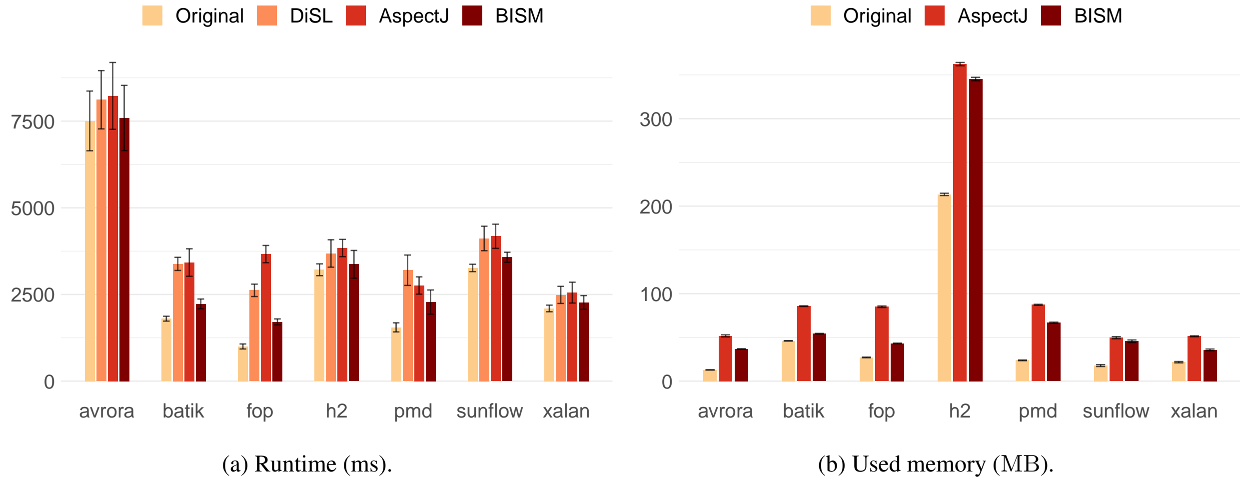
[11]The **HasNext** property specifies that a program should always call method `hasNext()` before calling method `next()` on an iterator. The **UnSafeIterator** property specifies that a collection should not be updated when an iterator associated with it

Table 3: Number of events generated by the financial transaction system for each monitored property.

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

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.
- In AspectJ, we use the call pointcut, type pattern matching, and joinpoint static information to capture method calls.



(a) Runtime (ms).

(b) Used memory (MB).

Figure 9: DaCapo loadtime-time instrumentation.

Load-time evaluation. Figure 9 reports the runtime for the benchmarks. BISM shows better performance over DiSL and AspectJ in 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. We mention that we did not measure the used memory for DiSL since it performs instrumentation on a separate JVM process.

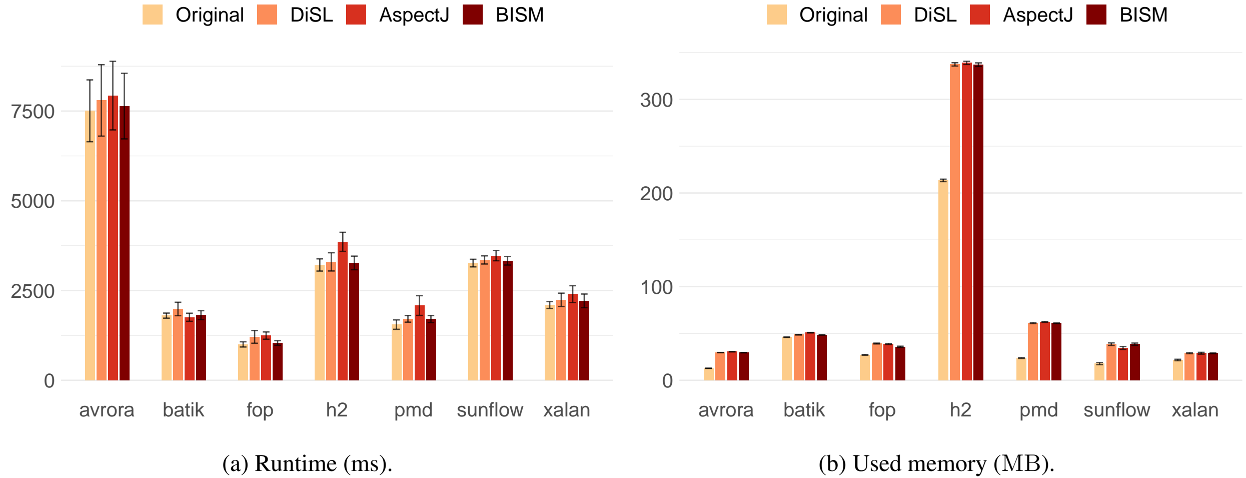
Build-time evaluation. We replace the original classes in the benchmarks with statically instrumented classes from each tool. Figure 10 reports the runtime and used memory of the benchmarks. BISM shows less runtime overhead in all benchmarks, except for batik where AspectJ emits fewer events. BISM also 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.

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 have been developed. In the following, we compare BISM to other Java instrumentation tools. Nevertheless, there are several tools to instrument

is being used. The **SafeSyncMap** property specifies that a map should not be updated when an iterator associated with it is being used.



(a) Runtime (ms).

(b) Used memory (MB).

Figure 10: DaCapo build-time 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 events emitted, (#) for BISM and DiSL, and AspectJ separately.

Benchmark	Scope	Instr.	Ref.	BISM		DiSL		AspectJ		Events (M)	
			kB	kB	Ovh.%	kB	Ovh.%	kB	Ovh.%	#	AspectJ
avrora	1,550	35	257	264	2.72	270	5.06	345	34.24	2.5	2.5
batik	2,689	136	1,544	1,572	1.81	1,588	2.85	1,692	9.59	0.5	0.4
fop	1,336	172	1,784	1,808	1.35	1,876	5.16	2,267	27.07	1.6	1.5
h2	472	61	694	704	1.44	720	3.75	956	37.75	28	28
pmd	721	90	756	774	2.38	794	5.03	980	29.63	6.6	6.3
sunflow	221	8	69	71	2.90	74	7.25	85	23.19	3.9	2.6
xalan	661	9	100	101	1.00	103	3.00	116	16.00	1	1

programs in different programming languages. For instance, to instrument C/C++ programs AspectC/C++ [20, 21] (high-level) and LLVM [22] (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 main features important 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 AOP Paradigm.

BCEL [7] 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 it requires proficiency in bytecode.

ASM [6] is a bytecode manipulation framework utilized by several tools, including BISM. ASM offers two APIs that can be used interchangeably to parse, load, and modify classes. However, to use ASM, a developer must deal with the

Table 5: Comparison of some of the main tools for instrumenting Java programs along 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 [7]	ASM [6]	Javassist [23]	CGLIB [24]	DiSL [8]	AspectJ [5]	BISM [12]
Provides Bytecode Visibility	✓	✓	✓	✓	✓	✗	✓
Allows Bytecode Insertion	✓	✓	✓	✓	✗	✗	✓
Requires no Bytecode Proficiency	✗	✗	✓✗	✓✗	✓	✓	✓
Provides high-Level Abstraction	✗	✓✗	✓✗	✓✗	✓	✓	✓
Follows AOP Paradigm	✗	✗	✗	✗	✓	✓	✓

low-level details of bytecode instructions and the JVM. BISM offers extended ASM compatibility and provides better abstraction with its aspect-oriented paradigm.

Javassist [23] is a class library for editing bytecodes 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 have knowledge about Java bytecode.

CGLIB [24] 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.

DiSL [8] 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 Section 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 [25], 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 [5] is the standard aspect-oriented programming [2] 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 Section 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 [26], a framework for 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 [27], 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.

Also, the composition is studied concerning the base program and a single aspect. In [28] and [29], 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 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 (Section 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 (Section 8.2 and Section 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.

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 believe that BISM can be used as an alternative to AspectJ and DiSL for lightweight and expressive runtime verification and even runtime enforcement (cf. [30, 31, 32]) thanks to its bytecode insertion capability, equivalent or better performance, and ease of use. The reported use cases also demonstrate BISM's 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 in 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 [31, 32] tools can be implemented.

References

[1] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018.

[2] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[3] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. In Christian Colombo and Martin Leucker, editors, *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, volume 11237 of *Lecture Notes in Computer Science*, pages 241–262. Springer, 2018.

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

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

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

[7] Apache Commons. BCEL (byte code engineering library). https://commons.apache.org/proper/commons-bcel. Accessed: 2020-06-18.

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

[9] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 169–190. ACM, 2006.

[10] Chukri Soueidi, Ali Kassem, and Yliès Falcone. BISM: bytecode-level instrumentation for software monitoring. In Jyotirmoy Deshmukh and Dejan Nickovic, editors, *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*, volume 12399 of *Lecture Notes in Computer Science*, pages 323–335. Springer, 2020.

[11] Giles Reger, Sylvain Hallé, and Yliès Falcone. Third international competition on runtime verification - CRV 2016. In Yliès Falcone and César Sánchez, editors, *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, volume 10012 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 2016.

[12] BISM: Bytecode-Level Instrumentation for Software Monitoring.

[13] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 2011.

[14] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 1978.

[15] A. Jefferson Offutt and Roland H. Untch. *Mutation 2000: Uniting the Orthogonal*, pages 34–44. Springer US, Boston, MA, 2001.

[16] T. Honglei, S. Wei, and Z. Yanan. The research on software metrics and software complexity metrics. In *2009 International Forum on Computer Science-Technology and Applications*, 2009.

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

[18] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 1976.

[19] Jerry Fitzpatrick. Applying the ABC metric to C, C++, and Java. pages 245–264, 01 2000.

[20] Yvonne Coady, Gregor Kiczales, Michael J. Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In A Min Tjoa and Volker Gruhn, editors, *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*, pages 88–98. ACM, 2001.

[21] O. Spinczyk, Daniel Lohmann, and M. Urban. AspectC++: An AOP extension for C. *Software Developer's Journal*, 01 2005.

[22] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[23] Shigeru Chiba. Load-time structural reflection in java. In Elisa Bertino, editor, *ECOOP 2000 - Object-Oriented Programming, 14th European Conference, Sophia Antipolis and Cannes, France, June 12-16, 2000, Proceedings*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer, 2000.

[24] Sam Berlin et al. CGLIB (byte code generation library). https://github.com/cglib/cglib. Accessed: 2021-05-21.

[25] Walter Binder, Philippe Moret, Éric Tanter, and Danilo Ansaloni. Polymorphic bytecode instrumentation. *Softw. Pract. Exp.*, 46(10):1351–1380, 2016.

[26] Rémi Douence, Pascal Fradet, and Mario Sûdholt. A framework for the detection and resolution of aspect interactions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2487:173–188, 2002.

[27] Fuminobu Takeyama and Shigeru Chiba. An advice for advice composition in AspectJ. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6144 LNCS:122–137, 2010.

[28] Wilke Havinga, I Nagy, and Lodewijk M J Bergmans. An Analysis of Aspect Composition Problems. *Proceedings of the Third European Workshop on Aspects in Software, 2006, Enschede, Netherlands*, (Technical Report IAI-TR-2006-6):1–8, 2006.

[29] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. *ACM International Conference Proceeding Series*, 208:85–95, 2007.

[30] Yliès Falcone. You should better enforce than verify. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2010.

[31] Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. Runtime failure prevention and reaction. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 103–134. Springer, 2018.

[32] Yliès Falcone and Srinivas Pinisetty. On the runtime enforcement of timed properties. In Bernd Finkbeiner and Leonardo Mariani, editors, *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11757 of *Lecture Notes in Computer Science*, pages 48–69. Springer, 2019.