DiSL: A Domain-Specific Language for Bytecode Instrumentation

Lukáš Marek
Charles University, Czech Republic
lukas.marek@d3s.mff.cuni.cz

Alex Villazón
Universidad Privada Boliviana, Bolivia
avillazon@upb.edu

Yudi Zheng
Shanghai Jiao Tong University, China
zheng.yudi@sjtu.edu.cn

Danilo Ansaloni Walter Binder
University of Lugano, Switzerland
{danilo.ansaloni, walter.binder}@usi.ch

Zhengwei Qi
Shanghai Jiao Tong University, China
qizhwei@sjtu.edu.cn

Abstract

Many dynamic analysis tools for programs written in managed languages such as Java rely on bytecode instrumentation. Tool development is often tedious because of the use of low-level bytecode manipulation libraries. While aspect-oriented programming (AOP) offers high-level abstractions to concisely express certain dynamic analyses, the join point model of mainstream AOP languages such as AspectJ is not well suited for many analysis tasks and the code generated by weavers in support of certain language features incurs high overhead. In this paper we introduce DiSL (domain-specific language for instrumentation), a new language especially designed for dynamic program analysis. DiSL offers an open join point model where any region of bytecodes can be a shadow, synthetic local variables for efficient data passing, efficient access to comprehensive static and dynamic context information, and weave-time execution of user-defined static analysis code. We demonstrate the benefits of DiSL with a case study, recasting an existing dynamic analysis tool originally implemented in AspectJ. We show that the DiSL version offers better code coverage, incurs significantly less overhead, and eases the integration of new analysis features that could not be expressed in AspectJ.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks

General Terms Languages, Measurement, Performance

Keywords Bytecode instrumentation, dynamic program analysis, aspect-oriented programming, JVM

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD'12, March 25–30, 2012, Potsdam, Germany.
Copyright © 2012 ACM 978-1-4503-1092-5/12/03. . . $10.00

1. Introduction

Dynamic program analysis tools support numerous software engineering tasks, including profiling, debugging, testing, program comprehension, and reverse engineering. Despite of the importance of dynamic analysis, prevailing techniques for building dynamic analysis tools are based on low-level abstractions that make tool development, maintenance, and customization tedious, error-prone, and hence expensive. For example, many dynamic analysis tools for the Java Virtual Machine (JVM) rely on bytecode instrumentation, supported by a variety of bytecode engineering libraries that offer low-level APIs resulting in verbose implementation code.

In an attempt to simplify the development of dynamic analysis tools, researchers have explored the use of aspect-oriented programming (AOP) languages, such as AspectJ [16]. Examples of aspect-based dynamic analysis tools are the DJProf profilers [20], the RacerAJ data-race detector [10], and the Senseo Eclipse plugin for augmenting static source code views with dynamic information [21]. However, as neither mainstream AOP languages nor the corresponding weavers have been designed to meet the requirements of dynamic program analysis, the success of using AOP for dynamic analysis remains limited. For example, in AspectJ, join points that are important for dynamic program analysis (e.g., the execution of bytecodes or basic blocks) are missing, access to reflective dynamic join point information is expensive, data passing between woven advice in local variables is not supported, and the mixing of low-level bytecode instrumentation and high-level AOP code is not foreseen.

In this paper, we introduce DiSL, a new domain-specific language for bytecode instrumentation. DiSL relies on AOP principles for concisely expressing efficient dynamic analysis tools. The language provides an open join point model defined by an extensible set of bytecode markers, efficient access to static and dynamic context information, optimized processing of method[1] arguments, and synthetic local vari-

[1] In this paper, "method" stands for "method or constructor".

ables for efficient data passing. While DiSL significantly raises the abstraction level when compared to prevailing bytecode manipulation libraries, it also exposes a low-level API to implement new bytecode markers. The DiSL weaver guarantees complete bytecode coverage to ensure that analysis results represent overall program execution. DiSL follows similar design principles as @J [8], an AOP language for dynamic analysis, which however lacks an open join point model and efficient access to method arguments.

Compared to high-level dynamic analysis frameworks such as RoadRunner [13] or jchord[2] that restrict the locations that can be instrumented, DiSL offers the developer fine-grained control over the inserted bytecode; that is, DiSL is not tailored for any specify dynamic analysis task, but provides constructs for concisely expressing any bytecode instrumentation. Instrumentation sites can be specified with a combination of bytecode markers, scoping expressions, and guards; guards represent static analyses executed at weave-time. Instrumentation code is provided in the form of snippets, that is, code templates that are instantiated for each selected instrumentation site and inlined. Snippets may access synthetic local variables to pass data from one instrumentation site to another. Snippets may access any static or dynamic context information; they may also process an arbitrary number of method arguments in a custom way.

The scientific contributions of this paper are twofold:

1. We present our design goals, the DiSL language constructs, and the implementation of the DiSL weaver.
2. We present a case study to illustrate the benefits of DiSL. We recast Senseo [21, 22] in DiSL; Senseo is an AspectJ-based profiling tool that supports various software maintenance tasks. In contrast to the former AspectJ implementation, the DiSL version of the tool features complete bytecode coverage, introduces significantly less overhead, and can be easily extended to collect additional dynamic metrics on the intra-procedural control flow.

This paper is structured as follows: Section 2 describes the design goals underlying DiSL. Section 3 gives a detailed overview of the DiSL language constructs. The software architecture of the DiSL weaver and its implementation are discussed in Section 4. Our case study is introduced in Section 5 and evaluated in Section 6. Section 7 discusses related work, Section 8 summarizes the strengths and limitations of DiSL, and Section 9 concludes.

2. Design of DiSL

Designing a good language for instrumentation-based dynamic program analysis is challenging, because we need to reconcile three conflicting design goals: (1) high expressiveness of the language, (2) a convenient, high-level programming model, and (3) high efficiency of the developed analysis tools. On the one hand, existing bytecode manipulation libraries meet the first and the third goal, but provide only low-level abstractions that make tool development cumbersome. On the other hand, mainstream AOP languages achieve the second goal, but lack expressiveness (e.g., lack of join points that would allow tracing the intra-procedural control flow) and suffer from inefficiencies (e.g., access to dynamic reflective join point information may require the allocation of unnecessary objects). The design of DiSL aims at bridging the gap between low-level bytecode manipulation frameworks and high-level AOP. Below, we motivate the main design choices underlying DiSL.

Open join point model. DiSL allows any region of bytecodes to be used as a join point, thus following an open join point model. That is, the set of supported join point *shadows* [15] is not hard-coded. To enable the definition of new join points, DiSL provides an extensible mechanism for marking user-defined bytecode regions (i.e., shadows).

Compatibility with Java and the JVM. DiSL is a domain-specific embedded language which has Java as its host language. DiSL instrumentations are implemented in Java, and annotations are used to express where programs are to be instrumented. Dynamic analysis tools written in DiSL can be compiled with any Java compiler and executed on any JVM.

Advice inlining and data passing in synthetic local variables. Advice in DiSL are expressed in the form of code *snippets* that are inlined, giving the developer fine-grained control over the inserted code. DiSL *instrumentations* (corresponding to aspects in AOP) describe where snippets are to be inserted into the base program. Thanks to inlining, snippets woven into the same method are able to efficiently communicate data through *synthetic local variables* [6].

Efficient access to complete static and dynamic context information. In DiSL, all static context information is exposed to the developer. This feature is similar to AspectJ's static reflective join point information (offering class and method properties), but exposes additional information at the basic block and bytecode level. DiSL also supports user-defined static analysis to compute further static context information at weave-time. In addition, DiSL provides a simple, yet powerful reflective API to gather dynamic context information which gives access to local variables and to the operand stack, supporting also efficient access to an arbitrary number of method arguments.

No support for around advice. Mainstream AOP languages support advice execution *before*, *after*, and *around* join points. Three common use cases of around advice are (1) passing data around a join point, (2) skipping a join point, and (3) executing a join point multiple times. As we assume that instrumentations do not alter the control flow in the base program, only the first use is relevant for us. However, for the first use case, the same behavior can be achieved with before and after advice using synthetic local variables [6].

[2] `http://code.google.com/p/jchord/`

Hence, DiSL only supports before and after advice, which helps keep the weaver simple.

Complete bytecode coverage. DiSL is designed for weaving with complete bytecode coverage. That is, the DiSL weaver ensures that all methods that have a bytecode representation can be woven, including methods in the standard Java class library. To this end, the DiSL weaver relies on implementation techniques developed in previous work [19].

3. Language Features

In this section we give an overview of the language features of DiSL. In Section 3.1 we introduce DiSL instrumentations specified in the form of snippets; markers determine where snippets are woven in the bytecode. The mechanism to control the inlining order of snippets is explained in Section 3.2. Synthetic local variables for efficiently passing data between woven snippets are presented in Section 3.3, and efficient access to thread-local variables is discussed in Section 3.4. In Section 3.5 we introduce static context to provide static reflective information, and we present the reflective API for obtaining dynamic context information in Section 3.6. In Section 3.7 we explain DiSL's support for method arguments processing. In Section 3.8 we introduce guards that enable the evaluation of conditionals at weave-time to decide whether a join point is to be captured, as well as a scoping construct to restrict weaving.

3.1 Instrumentations, Snippets, and Markers

DiSL *instrumentations* are Java classes. An instrumentation can only have *snippets* that are static methods annotated with `@Before`, `@After`, `@AfterReturning`, or `@AfterThrowing`. Snippets are defined as static methods, because their body is used as a template that is instantiated and inlined at the matching join points in the base program. Snippets do not return any value and must not throw any exception (that is not caught by a handler in the snippet).

Because of DiSL's open join point model, pointcuts are not hardcoded in the language but defined by an extensible library of *markers*. Markers are standard Java classes implementing a special interface for join point selection. DiSL provides a rich library of markers including those for method body, basic block, individual bytecode, and exception handler. In addition, the developer may extend existing markers or implement new markers from scratch.

The marker class is specified in the `marker` attribute in the snippet annotation. The weaver takes care of instantiating the selected marker, matching the corresponding join points, and weaving the snippets.

In addition to the predefined markers, DiSL offers join point extensibility by exposing the internal representation of method bodies to the developer, who has to implement code to *mark* the bytecode regions defining the shadows for the new join points.

3.2 Control of Snippet Order

It is common that several shadows coincide in the starting instruction, that is, several snippets may apply to the same join point. Similar to AspectJ's *advice precedence* resolution, DiSL provides a simple mechanism to control snippet ordering through the `order` attribute in the snippet annotation. The order is specified as a non-negative integer value. For `@Before`, snippets with higher order are inlined before snippets with lower order. For `@After`, `@AfterReturning`, and `@AfterThrowing`, snippets with lower order are inlined before snippets with higher order. Thus, the order indicates "how close" to the shadow the snippet shall be inlined.

3.3 Synthetic Local Variables

DiSL provides an efficient communication mechanism to pass arbitrary data between snippets. The mechanism relies on inlining so as to store the data in a local variable, which is therefore visible in the scope of the woven method body. DiSL provides the `@SyntheticLocal` annotation to specify the holder variable. Synthetic local variables must be declared as static fields and can be used in any snippet. The weaver takes care of translating the static field declared in the instrumentation into a local variable in each instrumented method, and of replacing the bytecodes that access the static field with bytecodes that access the introduced local variable. For details, we refer to [6].

3.4 Thread-local Variables

DiSL supports thread-local variables with the `@ThreadLocal` annotation. This mechanism extends `java.lang.Thread` by inserting the annotated field. While the inserted fields are instance fields, thread-local variables must be declared as static fields in the instrumentation class, similar to synthetic local variables. These fields must be initialized to the default value of the field's type.[3] The DiSL weaver translates all access to thread-local variables in snippets into bytecodes that access the corresponding field of the currently executing thread. An `inheritable` flag can be set in the `@ThreadLocal` annotation such that new threads "inherit" the value of a thread-local variable from the creating thread. Note that the standard Java class library offers classes with similar semantics (`java.lang.ThreadLocal` and `java.lang.InheritableThreadLocal`). However, accessing fields directly inserted into `java.lang.Thread` results in more efficient code.

3.5 Static Context Information

Accessing static context information is essential for dynamic analyses, for example, gathering information about

[3] During JVM bootstrapping, in general, inserted code cannot be executed because it may introduce class dependencies that can violate JVM assumptions concerning the class initialization order. Hence, threads created during bootstrapping could not initialize inserted thread-local fields in the beginning.

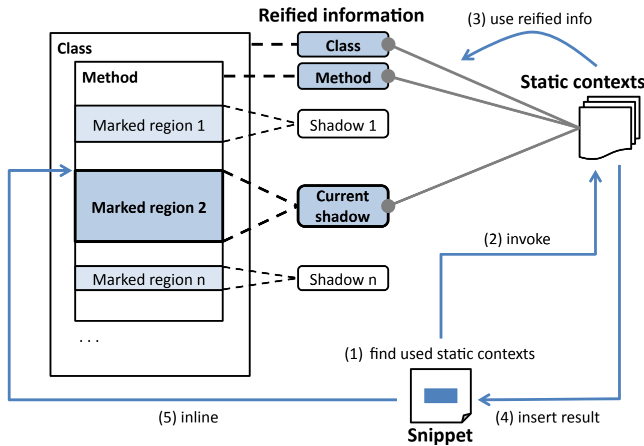

Figure 1. Gathering static context information at weave-time

```
public class CallingContextBBAnalysis {
  @ThreadLocal
  static CCTNode currentNode;

  @SyntheticLocal
  static CCTNode callerNode;

  @Before(marker = BodyMarker.class, order = 1)
  static void onMethodEntry(MethodStaticContext msc) {
    if ((callerNode = currentNode) == null)
      callerNode = CCTNode.getRoot();
    currentNode =
      callerNode.profileCall(msc.thisMethodFullName());
  }

  @After(marker = BodyMarker.class)
  static void onMethodCompletion() {
    currentNode = callerNode;
  }

  @Before(marker = BasicBlockMarker.class, order = 0)
  static void onBasicBlock(BasicBlockStaticContext bbsc) {
    currentNode.profileBB(bbsc.getBBIndex());
  }
}
```

Figure 2. Sample instrumentation for calling context-aware basic block profiling (class CCTNode is not shown)

the method, basic block, or bytecode instruction that is executed. Because of the open join point model of DiSL, there is no bound static part of a join point as in AspectJ. In DiSL, the programmer can gather reflective static information at weave-time by using various *static contexts*. DiSL provides a library of commonly used static contexts such as `MethodStaticContext`, `BasicBlockStaticContext`, and `BytecodeStaticContext`. The developer may also implement custom static context classes.

For every snippet, the programmer can specify any number of static contexts as argument. Each static context class implements the `StaticContext` interface and provides methods without argument that must return a value of a Java primitive type or a string. The reason for this restriction is that DiSL stores the results of static context methods directly in the constant pool of the woven class. Static contexts receive read-only access to the shadow containing the following reflective information: the class and method under instrumentation, the snippet, and the beginning and ending positions of the current shadow.

Figure 1 depicts the reflective approach for gathering static context information. After shadow marking according to the selected marker, the snippet is parsed to locate invocations to static context methods (step 1). Static contexts are then instantiated by the weaver and the corresponding methods are invoked for every shadow (step 2). Static context methods access the exposed reflective data to compute the static information to be returned (step 3). The weaver replaces the invocation of the static context methods in the snippet with bytecodes to access the computed static information (step 4). The snippet code is inlined before or after the matching shadows (step 5).

Figure 2 shows how static contexts are used in an instrumentation for calling context-aware basic block analysis. The goal is to help developers find hotspots in their programs taking both the inter- and intra-procedural control flow into account. The presented instrumentation collects statistics on basic block execution for each calling context.

For storing inter-procedural calling context information, a Calling Context Tree (CCT) [3] is used. For each thread, the current CCT node is kept in the thread-local variable `currentNode` that is updated upon method entry and completion (`onMethodEntry`(...) and `onMethodCompletion`() snippets using the `BodyMarker`). The synthetic local variable `callerNode` is used to store the CCT node corresponding to the caller. The `CCTNode.getRoot`() method returns the root node of the CCT. The method `profileCall`(...) takes a method identifier as argument and returns the corresponding callee node in the CCT. The method identifier is obtained from the `MethodStaticContext`; it is inserted as a string in the constant pool of the woven class.[4]

The `onBasicBlock`() snippet captures all basic block join points using the `BasicBlockMarker`. The idea is to count how many times each basic block is executed, so as to detect hot basic blocks. To this end, the snippet uses the `BasicBlockStaticContext` for gathering the index of the captured basic block. This value is used to increment the corresponding counter in the CCT node (not shown). Note that the `order` of the `@Before` snippets ensures that the initialization of the synthetic local variable `callerNode` and the update of the thread-local variable `currentNode` are done at the very beginning of the method body, before they are accessed in the first basic block.

3.6 Dynamic Context Information

Access to dynamic join point information (e.g., `getThis`(), `getTarget`(), and `getArgs`() in AspectJ) requires gathering data from *local variables* and from the *operand*

[4] This is similar to the use of `JoinPoint.StaticPart` in AspectJ. While AspectJ inserts static fields in the woven class to hold reflective static join point information, DiSL avoids structural modifications of the woven class.

```
public interface DynamicContext {
  <T> T getLocalVariableValue(int index,
                              Class<T> valueType);
  <T> T getStackValue(int distance, Class<T> valueType);
  Object getThis();
}
```

Figure 3. DynamicContext interface

```
public class ArrayAccessAnalysis {
  @Before(marker = BytecodeMarker.class, args = "aastore")
  static void beforeArrayStore(DynamicContext dc) {
    Object array = dc.getStackValue(2, Object.class);
    int index = dc.getStackValue(1, int.class);
    Object stored = dc.getStackValue(0, Object.class);
    Analysis.process(array, index, stored); // not shown
  }
}
```

Figure 4. Profiling array access

stack [15]. DiSL provides an API to explicitly access this information. Figure 3 shows the `DynamicContext` API which provides reflective information through the `getLocalVariableValue`(...) to access a local variable, `getStackValue`(...) to access a stack value, and `getThis`() returning this object or null in the case of a static method. Similar to static contexts, the `DynamicContext` can be passed to snippets as an argument. The programmer must provide the index and the type of the data to access. Note that the use of `DynamicContext` is not restricted to any particular marker. The developer must know how to access the correct data from local variables or from the operand stack. The weaver takes care of translating calls to the API methods into bytecode sequences to retrieve the desired values.

An example of the use of `DynamicContext` is access to the return value of a method, which is on top of the stack upon normal method completion. The programmer may implement an `@AfterReturning` snippet with the `BytecodeMarker` (for different `return` bytecodes) and use `getStackValue`(0, ...) to retrieve the return value. The index zero indicates the top of the stack.

The combination of `DynamicContext` with the `BytecodeMarker` provides a powerful mechanism to gather join point information for implementing dynamic analysis tools, such as memory profilers. For example, Figure 4 shows how to capture array accesses, which is not possible in AspectJ. The `beforeArrayStore`(...) snippet captures all objects being stored in arrays, where the element type is a reference type. The profiler can keep track which object has been stored at which position of an array. Before every `aastore` bytecode, the snippet gets the `array`, the `index`[5] where the element will be stored, and the object to be `stored` from the operand stack (at positions 2, 1, and 0, respectively). The `process`(...) method processes the collected information (not shown).

[5] The use of Java generics in the API results in autoboxing of primitive values (e.g., `index`) in the compiled snippet. The DiSL weaver removes the unnecessary boxing code before inlining.

```
public interface ArgumentProcessorContext {
  Object getReceiver(ArgumentProcessorMode mode);
  Object[] getArgs(ArgumentProcessorMode mode);
  void apply(Class<?> argumentProcessor,
             ArgumentProcessorMode mode);
}

public enum ArgumentProcessorMode {
  METHOD_ARGS, CALLSITE_ARGS
}
```

Figure 5. Argument processor API

3.7 Argument Processors

Method arguments are retrieved from local variables or, in the case of call sites, from the operand stack. DiSL's `DynamicContext` can be used to access these values when the argument index and type are known, which is not always the case. DiSL also provides a reflective mechanism, called *argument processor*, to process all arguments by their types.

The `ArgumentProcessorContext` interface (see Figure 5) can be used within snippets to access method arguments; it is to be passed to snippets as an argument, similar to static contexts or `DynamicContext`. Two modes can be specified, to process either arguments of the method where the snippet is inlined (`METHOD_ARGS`), or arguments of a method invocation (`CALLSITE_ARGS`). The `getReceiver`(...) method returns the receiver, or `null` for static methods. The `getArgs`(...) method returns all arguments in an object array, similar to `JoinPoint.getArgs`() in AspectJ for `execution` respectively `call` pointcuts. However, if the programmer needs to selectively access arguments, or does not want them to be wrapped in an object array (e.g., for performance reasons or to preserve the original type for arguments of primitive types), the API provides the `apply`(...) method, where the programmer can specify an argument processor class that handles the generation of code to access the arguments.

Argument processors are classes annotated with `@ArgumentProcessor`. At weave-time, DiSL checks which argument processor is selected in the snippet, and for each matching join point, generates the code to process the arguments according to their types.

Argument processors must implement static `void` methods, where the first parameter is required and additional (optional) parameters may be passed. The type of the first parameter selects the type of argument to be captured. The first parameter's type can only be `java.lang.Object` or a primitive type. For each argument of the woven method, the weaver checks whether the selected argument processor has a method where the first parameter type matches the current method argument type. In this case, the weaver generates the code to access the corresponding argument, which is eventually inlined within the snippet. As additional parameters, the argument processor method can take any static context, `DynamicContext`, or `ArgumentContext`. `ArgumentContext` is an interface to access argument type, argument position, and the total number of arguments.

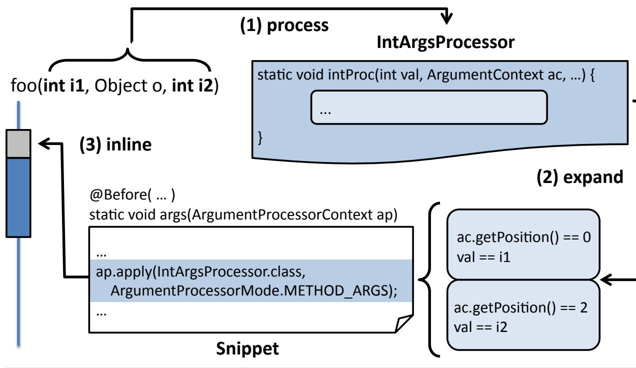

Figure 6. Processing of integer arguments

```
public class ArgumentAnalysis {
  @Before(marker = BodyMarker.class,
          guard = MethodReturnsRef.class)
  static void onMethodEntry {
    ... // inlined only if the method returns an object
  }
}

public class MethodReturnsRef {
  @GuardMethod
  static boolean evalGuard(ReturnTypeStaticContext rtsc) {
    return !rtsc.isPrimitive();
  }
}
```

Figure 7. Snippet guard restricting weaving to methods that return objects

Figure 6 illustrates the weaving of a snippet before a join point in method `foo(...)`. In this example, the developer only wants to process arguments of type `int`. For method `foo(...)`, only two of the arguments will match the `intProc(...)` processor method (`i1` and `i2`). First, the weaver finds out which argument processor and mode should be applied to the snippet (step 1). Then, the invocation to `apply(...)` in the snippet is replaced with the expanded method bodies of the processor for each matching argument (step 2). In the example, the generated code will give access to the two integer arguments, i.e., the snippet will contain expanded processor code to access the values `i1` and `i2`. Finally, the expanded snippet is inlined (step 3). For `METHOD_ARGS`, the generated code retrieves the arguments from local variables; for `CALLSITE_ARGS`, the arguments are taken from the operand stack. The use of `CALLSITE_ARGS` throws a weave-time error if the snippet is not woven before a method invocation bytecode.

There are several advantages of using argument processors compared to, for example, `JoinPoint.getArgs()` in AspectJ. Firstly, there is no need for creating objects that hold dynamic join point information. DiSL efficiently takes the correct values directly from local variables or from the stack. Secondly, argument types are preserved. The values of primitive types are not boxed as in AspectJ. Finally, it is straightforward to apply argument processors to a subset of arguments, without requiring complex pointcuts to be written. We will illustrate these advantages in more detail with our case study and evaluation in Sections 5 and 6.

3.8 Guards and Scope

DiSL provides two complementary mechanisms for restricting the application of snippets. The first one, *guard*, is based on weave-time evaluation of conditionals. The second one, *scope*, is based on method signature matching.

Guards allow us to evaluate complex weave-time restrictions for individual join points. A guard has to implement a static method annotated with `@GuardMethod`. The guard method may take any number of static contexts as arguments. The guard method returns a boolean value indicating whether the current joint point is to be instrumented. Static contexts can be used to expose reflective weave-time information to the guard. The guard has to be specified with the `guard` attribute of the snippet annotation.

In contrast to AspectJ's `if` pointcut, the evaluation of guards is done for each join point at weave-time. This avoids runtime overhead due to the evaluation of statically known conditionals. To illustrate this point, let's consider the example shown in Figure 7. The programmer wants to restrict weaving only to methods returning objects; methods returning values of primitive types (or `void`, which we consider a primitive type here) shall not be woven. The `evalGuard(...)` method of the `MethodReturnsRef` guard uses `ReturnTypeStaticContext` to determine whether the return type of the instrumented method is primitive. Because this evaluation is performed at weave-time, the `onMethodEntry(...)` snippet will be inlined only in methods that return objects.

Another interesting example of weave-time conditional evaluation is the use of data flow analysis within guards. This feature helps avoid inlining snippets that would otherwise access uninitialized objects (passing an uninitialized object to another method as argument would be illegal and cause a verification failure). For example, the programmer may capture all `putfield` bytecodes in constructors, where the target is a properly initialized object. Consequently, `putfield` bytecodes that write to the object under initialization before invocation of the superclass constructor will not be captured.

Even though guards are expressive, in many common cases, a more concise scoping expression is sufficient. In DiSL, scope is a simplified signature pattern matching pointcut designator. The `scope` attribute of the snippet annotation specifies which methods shall be instrumented. Scope expressions specify method, class, or package names and may contain wildcards (e.g., `scope = "* java.io.*(..)"`). Typically, scope evaluation is faster than guard evaluation, as it is done only once for each method. In contrary, a guard has to be invoked (using reflection) for each join point in the method. The best combination is the usage of scope expressions for fast method filtering and of guards for fine-grained joint point selection.

4. Implementation

DiSL is implemented in Java using the ASM[6] bytecode manipulation library in about 100 classes and 8000 lines of code. The DiSL weaver[7] runs on top of jBORAT[8], a lightweight toolkit providing support for instrumentation with complete bytecode coverage [19]. jBORAT uses two JVMs: an *instrumentation JVM* where bytecode instrumentation is performed and an *application JVM* that executes the instrumented application. This separation of the instrumentation logic from the instrumented application reduces perturbations in the application JVM (e.g., class loading and initialization triggered by jBORAT or by the DiSL weaver do not happen within the application JVM). DiSL simplifies deployment with scripts, hiding the complex JVM setup from the user.

Figure 8 gives an overview of the DiSL weaver running on top of jBORAT. During initialization, DiSL parses all instrumentation classes (step 1). Then it creates an internal representation for snippets and initializes the used markers, guards, static contexts, and argument processors. When DiSL receives a class from jBORAT (step 2), the weaving process starts with the snippet selection. The selection is done in two phases, starting with scope matching (step 3) and followed by shadow creation and selection. Shadows are created using the markers associated with the snippets selected in the previous scope matching phase. Shadows are evaluated by guards and only snippets with at least one valid shadow are selected (step 4). At this point, all snippets that will be used for weaving are known. Static contexts are used to compute the static information required by snippets (step 5). Argument processors are evaluated for snippets, and argument processor methods that match method arguments are selected (step 6). All the collected information is finally used for weaving (step 7). Argument processors are applied, and calls to static contexts are replaced with the computed static information. The weaver also generates the bytecodes to access dynamic context information. Finally, the woven class is emitted and passed back to jBORAT (step 8).

5. Case Study: Senseo

In this section, we illustrate the benefits of DiSL by recasting *Senseo* [21], a dynamic analysis tool for code comprehension and profiling. Senseo uses an aspect written in AspectJ for collecting calling context-sensitive dynamic information for each invoked method, including statistics on the runtime types of method arguments and return values, the number of method invocations, and the number of allocated objects. These metrics are visualized by an Eclipse plugin[9] that enriches the static source code views with the collected dynamic information. Senseo helps developers understand the dynamic behavior of applications and locate performance problems.

[6] `http://asm.ow2.org/`

[7] `http://disl.origo.ethz.ch/`

[8] jBORAT stands for Java Bytecode Overall Rewriting and Analysis Toolkit.

[9] `http://scg.unibe.ch/research/senseo`

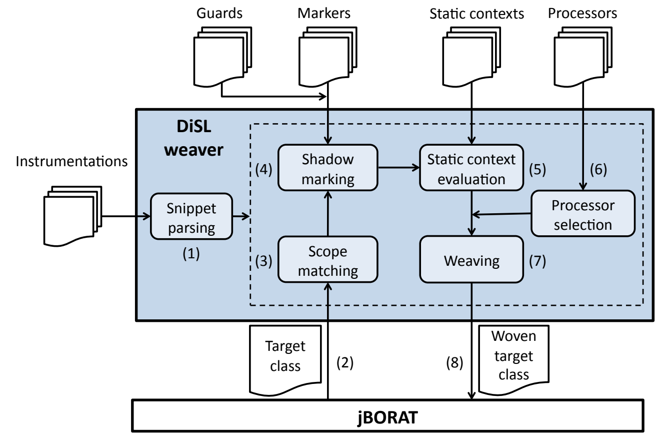

Figure 8. Overview of DiSL weaving process

The original version of Senseo has two main limitations: (1) lack of intra-procedural profiling and (2) high overhead for metrics collection. Both limitations stem from the use of AspectJ to express the instrumentation. Because of the absence of join points at the level of basic blocks, dynamic metrics on the intra-procedural control flow are missing, making it difficult for the developer to locate hot methods with complex intra-procedural control flow that are not invoked frequently. Moreover, access to dynamic join point information is inefficient due to the boxing of primitive values and because of the allocation of object arrays, notably for processing method arguments. For example, although only the first argument of method `paint(Object o, int x, int y)` could receive objects of different runtime types, the AspectJ implementation of Senseo collects the runtime types of all three arguments upon each invocation, because `JoinPoint.getArgs()` returns all arguments in a newly created object array, boxing values of primitive types.

Figure 9 shows the (simplified) DiSL instrumentation `Senseo2` that overcomes the limitations of the previous AspectJ implementation. To collect dynamic metrics for each calling context, the `onMethodEntry(...)` and `onMethodCompletion()` snippets reify the calling context in a similar way as explained in Section 3.5 (Figure 2). Each CCT node stores the dynamic information collected within the corresponding calling context, as explained below.

Number of method executions. The counting of method executions is subsumed in the `onMethodEntry(...)` snippet and performed in the `profileCall(...)` method by incrementing a counter. This information is used to compute the number of method calls for each calling context.

Number of allocated objects and arrays. To count the number of allocated objects and arrays, the `onAllocation()`

```
public class Senseo2 {
  @ThreadLocal
  static CCTNode currentNode;

  @SyntheticLocal
  static CCTNode callerNode;

  @Before(marker = BodyMarker.class, order = 1)
  static void onMethodEntry(MethodStaticContext msc,
      ArgumentProcessorContext proc) {
    if ((callerNode = currentNode) == null)
      callerNode = CCTNode.getRoot();
    currentNode =
      callerNode.profileCall(msc.thisMethodFullName());

    proc.apply(ReferenceProcessor.class,
               ProcessorMode.METHOD_ARGS);
  }

  @After(marker = BodyMarker.class, order = 2)
  static void onMethodCompletion() {
    currentNode = callerNode;
  }

  @AfterReturning(marker = BodyMarker.class, order = 1,
                  guard = MethodReturnsRef.class)
  static void onReturnRef(DynamicContext dc) {
    Object obj = dc.getStackValue(0, Object.class);
    currentNode.profileReturn(obj);
  }

  @AfterReturning(marker=BytecodeMarker.class, order=0,
        args = "new,newarray,anewarray,multianewarray")
  static void onAllocation() {
    currentNode.profileAllocation();
  }

  @Before(marker = BasicBlockMarker.class, order = 0)
  static void onBasicBlock(BasicBlockStaticContext bbsc){
    currentNode.profileBB(bbsc.getBBIndex());
  }
}

@ArgumentProcessor
public class ReferenceProcessor {
  static void objProc(Object obj, ArgumentContext ac) {
    Senseo2.currentNode.profileArgument(ac.getPosition(),
                                        obj);
  }
}
```

Figure 9. DiSL instrumentation for collecting runtime information for Senseo

snippet uses the `BytecodeMarker` to capture allocation bytecodes for both objects (`new`) and arrays (`newarray`, `anewarray`, and `multianewarray`). The `profileAllocation()` method updates an allocation counter in the current CCT node.

Runtime argument and return types. To collect runtime type information only for arguments of reference types, the `onMethodEntry(...)` snippet uses the argument processor `ReferenceProcessor`. Since this argument processor only defines the `objProc(...)` method to process arguments of reference types, all arguments with primitive types are automatically skipped. The `objProc(...)` method invokes the `profileArgument(...)` method of the current CCT node, passing the position of the argument and the reference.

For collecting runtime return types, the `onReturnRef(...)` snippet uses the `MethodReturnsRef` guard (see Figure 7 in Section 3.8) to ensure that the return type of a woven method is a reference type. Because the returned object reference is on top of the operand stack upon method completion, it is accessed with the `DynamicContext` API.

Basic-block metrics. As the execution of basic blocks cannot be captured with AspectJ, the following information is collected only by the DiSL version of Senseo. The `onBasicBlock(...)` snippet captures every basic block using the `BasicBlockMarker`; the `BasicBlockStaticContext` provides the index of the captured basic block (`getBBIndex()`). This allows us to keep track how many times a basic block is executed in each calling context.

Comparing different Senseo implementations. For a comparison of DiSL with low-level bytecode manipulation libraries and with AOP, it is interesting to consider the lines-of-code (LOC) used in the different implementations of the same tool. Hence, we implemented a third version of Senseo with the ASM bytecode manipulation library and compared the source code of the DiSL, AspectJ, and ASM versions. In contrast to the DiSL and ASM versions, the AspectJ version lacks basic block profiling, that is, it offers less functionality.

	DiSL	AspectJ	ASM
Physical lines-of-code	74	44	489
Logical lines-of-code	44	19	338

Table 1. Lines-of-code for three implementations of Senseo

Table 1 summarizes the physical and logical LOC metrics of the three implementations, considering only the code related to the actual instrumentation logic (and disregarding the Java code for analysis at runtime, which is common to all three implementations). Compared to ASM, the DiSL and AspectJ versions are significantly smaller, as the direct manipulation of bytecodes requires much more development effort than relying on the high-level pointcut/advice mechanism of AspectJ and DiSL. The higher LOC number of the DiSL implementation compared to the AspectJ version is mainly due to the separation of the code that is evaluated at weave-time (guards) from the instrumentation code (snippets). However, weave-time evaluation brings significant performance gains as we will show in Section 6.

In summary, our case study illustrates how DiSL enables the concise implementation of a practical dynamic analysis tool, thanks to DiSL's open join point model, efficient access to both static and dynamic context information, weave-time evaluation of conditionals, and argument processors. Dynamic analysis tools written in DiSL are much more concise than equivalent tools developed with bytecode manipulation libraries.

6. Performance Evaluation

In this section, we evaluate the runtime performance of the DiSL instrumentation presented in the Senseo case study.

	Reference	SenseoAJ		SenseoDiSL				Senseo2			
		application only		application only		full coverage		application only		full coverage	
	[s]	[s]	ovh.	[s]	ovh.	[s]	ovh.	[s]	ovh.	[s]	ovh.
avrora	5.11	30.96	6.06	12.61	2.47	12.41	2.43	13.66	2.67	14.62	2.86
batik	1.28	2.70	2.11	1.78	1.39	2.47	1.93	2.14	1.67	3.09	2.41
eclipse	16.16	152.92	9.46	70.73	4.38	81.52	5.04	152.41	9.43	163.36	10.11
fop	0.35	3.36	9.60	1.68	4.80	3.09	8.83	2.07	5.91	3.93	11.23
h2	5.84	63.25	10.83	25.27	4.33	31.78	5.44	29.55	5.06	41.81	7.16
jython	2.67	5.70	2.13	3.89	1.46	28.28	10.59	4.29	1.61	34.21	12.81
luindex	0.90	7.06	7.84	2.71	3.01	3.31	3.68	3.45	3.83	4.30	4.78
lusearch	1.98	13.09	6.61	5.49	2.77	6.57	3.32	6.19	3.13	8.85	4.47
pmd	2.05	10.09	4.92	5.10	2.49	7.60	3.71	6.54	3.19	10.31	5.03
sunflow	3.45	57.24	16.59	21.44	6.21	20.49	5.94	24.57	7.12	25.37	7.35
tomcat	1.97	4.46	2.26	3.16	1.60	6.70	3.40	3.87	1.96	9.32	4.73
tradebeans	5.56	71.48	12.86	30.76	5.53	76.43	13.75	42.90	7.72	117.40	21.12
tradesoap	6.77	25.40	3.75	12.80	1.89	53.60	7.92	17.30	2.56	76.12	11.24
xalan	1.11	20.39	18.37	8.15	7.34	11.38	10.25	10.08	9.08	17.33	15.61
geo. mean			6.47		3.09		5.26		3.91		7.19

Table 2. Execution times and overhead factors for `SenseoAJ`, `SenseoDiSL`, and `Senseo2`

First, we compare the previous AspectJ implementation with an equivalent DiSL instrumentation (i.e., without basic block metrics). In addition, we evaluate our DiSL instrumentation with full bytecode coverage, collecting also basic block metrics. Second, we explore the different sources of the measured overhead. Third, we investigate the differences in the collected profiles, considering the number of intercepted join points, when weaving only application code, respectively when weaving with full bytecode coverage. Fourth, we study weaving time and overall class loading latency due to jBORAT and DiSL.

For our measurements, both the DiSL weaver and the AspectJ weaver run on top of jBORAT. This ensures exactly the same weaving coverage for application code (otherwise, the AspectJ load-time weaver would exclude some application classes from weaving). Both the instrumentation JVM and the application JVM run on the same host. We use the benchmarks in the DaCapo suite (dacapo-9.12-bach)[10] as base programs in our evaluation. All measurements correspond to the median of 15 benchmark runs within the same application JVM. The measurement machine is an Intel Core2 Quad Q9650 (3.0 GHz, 8 GB RAM) that runs Ubuntu GNU/Linux 10.04 64-bit. We use AspectJ 1.6.11[11], DiSL pre-release version 0.9, and Oracle's JDK 1.6.0_27 Hotspot Server VM (64-bit) with 7 GB maximum heap size.

Table 2 reports the runtime overhead for the original AspectJ version of Senseo (`SenseoAJ`), for the equivalent instrumentation in DiSL, that is, without basic block metrics (`SenseoDiSL`), and for the DiSL instrumentation including basic block metrics (`Senseo2`). On average (geometric mean for DaCapo), the overhead factor introduced by `SenseoAJ` is 6.47, while for `SenseoDiSL`, with the same code coverage, the overhead is only a factor of 3.09. With full bytecode coverage, the average overhead of `SenseoDiSL` is a factor of 5.26; surprisingly, the overhead is still lower than for `SenseoAJ` covering only application code. Finally, the average overhead introduced by `Senseo2` is a factor of 7.19.

[10] `http://www.dacapobench.org/`

[11] `http://eclipse.org/aspectj/`

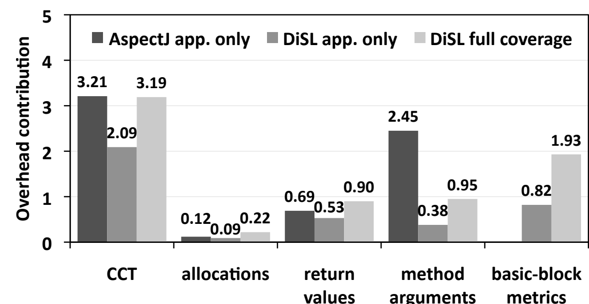

Figure 10. Contributions to the average overhead factor for different versions of Senseo

	application only	full coverage	increase [%]
Method bodies	5.60E+09	8.84E+09	57.75
Methods returning a ref.	1.76E+08	3.44E+08	95.28
Methods with ref. arg.	1.78E+09	2.57E+09	44.56
Object and array alloc.	1.14E+09	2.00E+09	76.11
Basic blocks	2.21E+10	3.34E+10	51.26

Table 3. Total number of intercepted join points for a single iteration of the whole DaCapo suite

Figure 10 quantifies the different overhead contributions. For CCT reification, the DiSL implementation benefits from efficient access to static context information, from data passing in synthetic local variables, and from the use of an `@ThreadLocal` variable (compared to a `java.lang.ThreadLocal` variable in `SenseoAJ`). The overheads for capturing allocations and runtime types of return values are relatively small for both implementations. The biggest difference between the two implementations is observed for the processing of method arguments; the DiSL instrumentation leverages an argument processor, whereas the AspectJ implementation relies on `JoinPoint.getArgs()`. As shown in Figure 10, argument processing in the AspectJ version introduces more than 6 times the overhead of the equivalent DiSL instrumentation.

Table 3 summarizes the number of intercepted join points for a single iteration of each considered benchmark, weaving only application code, respectively weaving with full byte-

	SenseoAJ	SenseoDiSL		Senseo2	
	app. only	app. only	full cov.	app. only	full cov.
Weaving [s]	54.97	43.28	134.17	65.61	174.73
Latency [s]	66.42	53.86	155.25	75.06	213.71

Table 4. Total weaving time and latency for a single iteration of the whole DaCapo suite

code coverage. For all kinds of join points, full bytecode coverage results in an increase of 45–95% in the number of intercepted join points. These results confirm that supporting weaving with full bytecode coverage is essential in the context of dynamic program analysis.

Finally, we compare the total time required to weave the complete benchmark suite. Table 4 reports (a) the total weaving time measured in the instrumentation JVM, and (b) the total weaving latency observed by the application JVM. This allows us to know the latency introduced by jBORAT. Overall, for application only, `SenseoAJ` is woven in 54.97s, whereas `SenseoDiSL` requires only 43.28s. The DiSL weaver outperforms the AspectJ weaver by a factor 1.27. With full coverage, `SenseoDiSL` requires 134.17s, and adding basic block metrics with `Senseo2` increases the weaving time to 65.61s for application code, and to 174.73s with full coverage. The latency contribution of jBORAT is between 14% and 24%, due to client-server communication.

Our evaluation confirms that DiSL enables the development of efficient dynamic analysis tools, which often cannot be achieved with general-purpose AOP languages. For our case study, the DiSL instrumentation reduces the overhead by more than factor 2 in comparison with the previous AspectJ version. Even with full bytecode coverage, the DiSL instrumentation still outperforms the AspectJ version.

7. Related Work

In previous work, we presented @J [8], a Java annotation-based AOP language for simplifying dynamic analysis. Similar to DiSL, @J uses snippet inlining and provides constructs for basic block analysis. However, @J lacks the open join point model of DiSL (i.e., @J does not support custom join point definitions), reflective access to weave-time information, and support for efficient access to reflective dynamic join point information (i.e., @J lacks argument processors). @J supports staged advice where weave-time evaluation of advice yields runtime residues that are woven. While this feature can be used to emulate guards in DiSL, it requires the use of additional synthetic local variables and more complex composition of snippets.

In [7] we discussed some early ideas on a high-level declarative domain-specific aspect language (DSAL) for dynamic analysis. DiSL provides all necessary language constructs to express the dynamic analyses that could be specified in the DSAL. That is, in the future, DiSL can serve as an intermediate language to which the higher-level DSAL programs are compiled.

High-level dynamic analysis frameworks such as RoadRunner [13] or jchord[12] ease composition of a set of common dynamic analyses. In contrast, DiSL is not tailored for any specify dynamic analysis task and offers the developer fine-grained control over the inserted bytecode.

The use of AOP for dynamic analysis [10, 20–22] has revealed some limitations in general-purpose AOP languages for that particular domain. In [1], a meta-aspect protocol (MAP) for dynamic analysis is proposed to overcome these limitations. Similar to our approach, the authors propose a flexible join point model where shadows are accessible in advice. Code snippets are used to inject callbacks to advice. MAP uses a meta object to reify context at runtime. While MAP allows fast prototyping of dynamic analyses, it does not focus on high efficiency of the developed analysis tools. In contrast, DiSL avoids any indirections to efficiently access static and dynamic context information.

The AspectBench Compiler (*abc*) [5] eases the implementation of AspectJ extensions. As intermediate representation, *abc* uses Jimple to define shadows. Jimple has no information where blocks, statements and control structures start and end, thus requiring extensions to support new pointcuts for dynamic analysis. In contrast, DiSL provides an extensible library of markers without requiring extensions of the intermediate representation.

Prevailing AspectJ weavers lack support for embedding custom static analysis in the weaving process. In [18] compile-time statically executable advice is proposed, which is similar to static context in DiSL. SCoPE [4] is an AspectJ extension that allows analysis-based conditional pointcuts. However, advice code together with the evaluated conditional is always inserted, relying on the just-in-time compiler to remove dead code. DiSL's guards together with static context allows weave-time conditional evaluation and can prevent the insertion of dead code.

In [2], the notion of region pointcut is introduced. Because a region pointcut potentially refers to several combined but spread join points, an external object shared between the join points holds the values to be passed between them. DiSL's markers provide a similar mechanism, and synthetic local variables help avoid passing data through an external object. In addition, region pointcuts are implicitly bound to the block structure of the program. In contrast, DiSL allows arbitrary regions to be marked.

Javassist [11] is a load-time bytecode manipulation library allowing definition of classes at runtime. The API allows two different levels of abstraction: source-level and bytecode-level. In particular, the source-level abstraction does not require any knowledge of the Java bytecode structure and allows insertion of code fragments given as source text. Compared to DiSL, Javassist does not follow a pointcut/advice model and does not provide built-in support for synthetic local variables.

[12] `http://code.google.com/p/jchord/`

Josh [12] is an AspectJ-like language that allows developers to define domain-specific extensions to the pointcut language. Similar to guards, Josh provides static pointcut designators that can access reflective static information at weave-time. However, the join point model of Josh does not include arbitrary bytecodes and basic blocks as in DiSL.

The approach described in [17] enables customized pointcuts that are partially evaluated at weave-time. It uses a declarative language to synthesize shadows. Because only a subset of bytecodes is converted to the declarative language, it is not possible to define basic block pointcuts as in DiSL.

Steamloom [9, 14] provides AOP support at the JVM level and improves performance of advice execution by optimizing dynamic pointcut evaluation. In DiSL, performance gains stem from static contexts combined with efficient access to dynamic context information. No JVM support is needed.

8. Discussion

In this section we discuss the strengths and limitations of DiSL for implementing dynamic analysis tools, comparing DiSL with the mainstream AOP language AspectJ [16] and with the low-level bytecode manipulation library ASM.

Expressiveness. AspectJ lacks certain join points that are important for some dynamic analysis tasks (e.g., bytecode-level and basic block-level join points). Thus, it is not possible to implement analysis tools that trace the intra-procedural control flow. In DiSL, any bytecode region can be a shadow, thanks to the support for custom markers. Likewise, with ASM, any bytecode location can be instrumented.

In AspectJ, the programmer has no control over the inserted bytecode. The AspectJ weaver inserts invocations to advice methods; inlining of advice is not foreseen. In contrast, the DiSL programmer writes snippets that are always inlined. If desired, it is trivial to mimic the behavior of the AspectJ weaver by writing snippet code that invokes "advice" methods. Still, if DiSL code is written in Java and compiled with a Java compiler, the snippets cannot contain arbitrary bytecode sequences. For example, it is not possible to write a snippet in Java that yields a single `dup` bytecode when inlined. Using ASM, there are no restrictions concerning the inserted bytecode.

Level of abstraction. In comparison with AspectJ, DiSL offers a lower abstraction level. The DiSL programmer needs to be aware of bytecode semantics, whereas AspectJ does not expose any bytecode-level details to the programmer. Nonetheless, DiSL relieves the developer from dealing with low-level bytecode manipulations such as producing specific bytecode sequences, introducing local variables, copying data from the operand stack, etc. Using ASM, the programmer also needs to deal with such low-level details, resulting in verbose tool implementations.

Compliance of the generated bytecode with the JVM specification. Weaving any aspect written in AspectJ results in valid bytecode that passes verification. In contrast, woven DiSL code may fail bytecode verification; it is up to the programmer to ensure that the inserted code is valid. For instance, synthetic local variables must be initialized before they are read, and the stack locations and local variables accessed through `DynamicContext` must be valid. Similarly, bytecode instrumented with tools written in ASM may fail verification.

While it is usually desirable that woven code passes verification, violating certain constraints on bytecode sometimes simplifies analysis tasks. For example, if the analysis needs to keep track of objects that are currently being initialized by a thread, the programmer may want to store uninitialized objects in a data structure on the heap, although the resulting bytecode would be illegal. Nonetheless, the analysis can be successfully executed by explicitly disabling bytecode verification. With AspectJ, such tricks are not possible.

Interference of inserted code with the base program. With ASM, local variables or data on the operand stack belonging to the base program may be unintentionally altered by inserted code. In contrast, AspectJ and DiSL guarantee that instrumentations cannot modify local variables or stack locations of the base program.

Bytecode coverage. For many analysis tasks, it is essential that the overall execution of the base program can be analyzed. However, prevailing AspectJ weavers do not support weaving the Java class library. In contrast, DiSL has been designed for weaving with complete bytecode coverage, which does not introduce any extra effort for the developer. With ASM, it is possible to develop tools that support complete bytecode coverage. However, the ASM programmer has to manually deal with the intricacies of bootstrapping the JVM with a modified class library and preventing infinite regression when inserted bytecode calls methods in the instrumented class library.

9. Conclusion

In this paper we presented DiSL, a new domain-specific language for bytecode instrumentation. The language is embedded in Java and makes use of annotations. DiSL allows the programmer to express a wide range of dynamic program analysis tasks in a concise manner. DiSL has been inspired by the pointcut/advice mechanism of mainstream AOP languages such as AspectJ. On the one hand, DiSL omits certain AOP language features that are not needed for expressing instrumentations (e.g., around advice and explicit structural modifications of classes). On the other hand, DiSL offers an open join point model, synthetic local variables, comprehensive and efficient access to static and dynamic context information, and support for weave-time execution of static analyses. These language features allow expressing bytecode transformations in the form of code snippets that are in-

lined before or after bytecode shadows as indicated by (custom) markers, if user-defined constraints specified as guards are satisfied. As case study, we recasted the dynamic analysis tool Senseo in DiSL and compared it with the previous implementation in AspectJ. In contrast to the AspectJ version, the DiSL implementation ensures complete bytecode coverage, reduces overhead, and allows us to gather additional intra-procedural execution statistics.

In an ongoing research project, we are working on advanced static checkers for DiSL instrumentations to help detect errors before weaving, on partial evaluation of instantiated snippets before inlining, and on general techniques to split overlong methods that exceed the maximum method size imposed by the JVM. In addition, we are exploring the use of higher-level, declarative domain-specific languages for dynamic program analysis. We plan to compile such higher-level languages to DiSL, which will serve us as a convenient intermediate language.

Acknowledgments

The research presented here was conducted while L. Marek, A. Villazón, and Y. Zheng were with the University of Lugano. It was supported by the Scientific Exchange Programme NMS–CH (project code 10.165), by a Sino-Swiss Science and Technology Cooperation (SSSTC) Exchange Grant (project no. EG26–032010) and Institutional Partnership (project no. IP04–092010), by the Swiss National Science Foundation (project CRSII2_136225), and by the Czech Science Foundation (project GACR P202/10/J042). The authors thank Aibek Sarimbekov and Achille Peternier for their help with jBORAT, and Andreas Sewe for testing DiSL and providing detailed feedback.

References

[1] M. Achenbach and K. Ostermann. A meta-aspect protocol for developing dynamic analyses. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 153–167. Springer-Verlag, 2010.

[2] S. Akai, S. Chiba, and M. Nishizawa. Region pointcut for AspectJ. In *ACP4IS '09: Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 43–48. ACM, 2009.

[3] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96. ACM, 1997.

[4] T. Aotani and H. Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 161–172. ACM, 2007.

[5] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98. ACM, 2005.

[6] W. Binder, D. Ansaloni, A. Villazón, and P. Moret. Flexible and efficient profiling with aspect-oriented programming. *Concurrency and Computation: Practice and Experience*, 23(15):1749–1773, 2011.

[7] W. Binder, P. Moret, D. Ansaloni, A. Sarimbekov, A. Yokokawa, and E. Tanter. Towards a domain-specific aspect language for dynamic program analysis: position paper. In *Proceedings of the sixth annual workshop on Domain-specific aspect languages*, DSAL '11, pages 9–11. ACM, 2011.

[8] W. Binder, A. Villazón, D. Ansaloni, and P. Moret. @J - Towards rapid development of dynamic analysis tools for the Java Virtual Machine. In *VMIL '09: Proceedings of the 3th Workshop on Virtual Machines and Intermediate Languages*, pages 1–9. ACM, 2009.

[9] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92. ACM, 2004.

[10] E. Bodden and K. Havelund. Aspect-oriented Race Detection in Java. *IEEE Transactions on Software Engineering*, 36(4):509–527, 2010.

[11] S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer Verlag, Cannes, France, June 2000.

[12] S. Chiba and K. Nakagawa. Josh: An open AspectJ-like language. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 102–111. ACM, 2004.

[13] C. Flanagan and S. N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 1–8. ACM, 2010.

[14] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An execution layer for aspect-oriented programming languages. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 142–152. ACM, 2005.

[15] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35. ACM, 2004.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353. Springer-Verlag, 2001.

[17] K. Klose, K. Ostermann, and M. Leuschel. Partial evaluation of pointcuts. In *Practical Aspects of Declarative Languages*, volume 4354 of *Lecture Notes in Computer Science*, pages 320–334. Springer-Verlag, 2007.

[18] K. Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: Checking the law of Demeter with AspectJ. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, AOSD '03, pages 40–49. ACM, 2003.

[19] P. Moret, W. Binder, and É. Tanter. Polymorphic bytecode instrumentation. In *AOSD '11: Proceedings of the 10th International Conference on Aspect-Oriented Software Development*, pages 129–140. ACM, Mar. 2011.

[20] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, June 2007.

[21] D. Röthlisberger, M. Härry, W. Binder, P. Moret, D. Ansaloni, A. Villazón, and O. Nierstrasz. Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks. *IEEE Transactions on Software Engineering*, PrePrint, 2011.

[22] D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting static source views in IDEs with dynamic metrics. In *ICSM '09: Proceedings of the 25th IEEE International Conference on Software Maintenance*, pages 253–262, Edmonton, Alberta, Canada, 2009. IEEE Computer Society.