Introduction to Dynamic Program Analysis with DiSL

Lukáš Marek[a], Yudi Zheng[b], Danilo Ansaloni[b], Lubomír Bulej[b], Aibek Sarimbekov[b], Walter Binder[b], Petr Tůma[a]

[a] *Charles University, Czech Republic*
[b] *University of Lugano, Switzerland*

Abstract

Dynamic program analysis (DPA) tools assist in many software engineering and development tasks, including profiling, program comprehension, and performance model construction and calibration. On the Java platform, many DPA tools are implemented either using aspect-oriented programming (AOP), or rely on bytecode instrumentation to modify the base program code. The pointcut/advice model found in AOP enables rapid tool development, but does not allow expressing certain instrumentations due to limitations of mainstream AOP languages—developers thus use bytecode manipulation to gain more expressiveness and performance. However, while the existing bytecode manipulation libraries handle some low-level details, they still make tool development tedious and error-prone. Targeting this issue, we provide the first complete presentation of DiSL, an open-source instrumentation framework that reconciles the conciseness of the AOP pointcut/advice model and the expressiveness and performance achievable with bytecode manipulation libraries. Specifically, we extend our previous work to provide an overview of the DiSL architecture, advanced features, and the programming model. We also include case studies illustrating successful deployment of DiSL-based DPA tools.

Keywords:
dynamic program analysis, bytecode instrumentation, aspect-oriented programming, domain-specific languages, Java Virtual Machine

Email addresses: `lukas.marek@d3s.mff.cuni.cz` (Lukáš Marek), `yudi.zheng@usi.ch` (Yudi Zheng), `danilo.ansaloni@usi.ch` (Danilo Ansaloni), `lubomir.bulej@usi.ch` (Lubomír Bulej), `aibek.sarimbekov@usi.ch` (Aibek Sarimbekov), `walter.binder@usi.ch` (Walter Binder), `petr.tuma@d3s.mff.cuni.cz` (Petr Tůma)

1. Introduction

With the growing complexity of software systems, there is an increased need for tools that allow developers and software engineers to gain insight into the dynamics and runtime behavior of those systems during execution. Such insight is difficult to obtain from static analysis of the source code, because the runtime behavior depends on many other factors, including program inputs, concurrency, scheduling decisions, and availability of resources.

Software developers therefore use various *dynamic program analysis* (DPA) tools, that observe a system in execution and distill additional information from its runtime behavior. The existing DPA tools can aid in a variety of tasks, including profiling [1, 2], debugging [3, 4, 5, 6], and program comprehension [7, 8], with increasingly sophisticated tools being introduced by the research community.

In the context of model-driven engineering (MDE), and specifically the area of model-based performance prediction and engineering, dynamic analyses can aid in automating construction and calibration of performance models. While there is a long-lasting trend towards deriving software performance models from other models created during development [9], a well-designed dynamic analysis can aid in construction of performance models for existing software and runtime platforms. Viewed from the perspective of aspect-oriented modeling approaches [10], dynamic analyses related to runtime performance monitoring can be considered crosscutting concerns, and represented as model aspects [11, 12] intended for composition with primary models.

The construction of DPA tools is difficult, in part because of the need to rewrite the base program code to capture occurrences of important events in the base program execution. On the Java platform, *bytecode instrumentation* is the prevailing technique used by existing DPA tools to modify the base program code. Libraries such as ASM [13] and BCEL [14] are often used to manipulate the bytecode, raising the level of abstraction to the level of classes, methods, and sequences of bytecode instructions, and relieving developers of the lowest-level details, such as handling the Java class files.

However, even with the bytecode manipulation libraries, implementing the instrumentation for a DPA tool is error-prone, requires advanced developer expertise, and results in code that is verbose, complex, and difficult to maintain. While frameworks such as Soot [15], Shrike [16], or Javassist [17], raise the level of abstraction further, they often target more general code transformation or optimization tasks in their design, which does not necessarily help in the instrumentation development.

Some researchers and authors of various DPA tools have thus turned to aspect-oriented programming (AOP) [18], which offers a convenient, high-level abstraction over predefined points in program execution (join points) and allows inserting code (advice) at a declaratively specified set of join points (pointcuts). This pointcut/advice model allows expressing certain instrumentations in a very concise manner, thus greatly simplifying the instrumentation development. Tools like the DJProf profiler [19], the RacerAJ data race detector [20], or the Senseo Eclipse plugin for augmenting static source views with dynamic metrics [8], all implemented using AspectJ [18], are examples of a successful application of this approach.

Despite the convenience of the AOP-based programming model, mainstream AOP languages such as AspectJ only provide a limited selection of join point types. The resulting lack of flexibility and expressiveness then makes many relevant instrumentations impossible to implement, forcing researchers back to using low-level bytecode manipulation libraries. Moreover, with AOP being primarily designed for purposes other than instrumentation, the high-level programming model and language features provided by AOP are often expensive in terms of runtime overhead [21].

To reconcile the convenience of the pointcut/advice model found in AOP and the expressiveness and performance attainable by using the low-level bytecode manipulation libraries, we have previously presented DiSL [22, 23, 24], an open-source framework that enables rapid development of efficient instrumentations for Java-based DPA tools. To raise the level of abstraction, DiSL adopts the AOP-based pointcut/advice model, which allows one to express instrumentations in a very concise manner, similar to AOP aspects. To retain the flexibility of low-level bytecode manipulation libraries, DiSL features an *open join-point model*, which allows any region of bytecodes to represent a join point. To achieve the performance attainable with low-level libraries, DiSL provides specialized features that provide constant-time access to static information related to an instrumentation site, which is computed at weave time, and features that allow caching and passing data between inserted code in different places. These features enable implementation of efficient instrumentations, without incurring the overhead caused by having to resort to very high-level, but costly, AOP features.

In addition, DiSL supports *complete bytecode coverage* [25] and mostly avoids[1] structural modifications of classes (i.e., adding methods and fields) that would be visible through the Java reflection API and could break the base program.

The general contribution of this paper is in providing the first complete presentation of the DiSL framework that serves as introduction to dynamic program analysis with DiSL. In previous work, we presented the design and the basic features of the DiSL framework [22, 23, 24]. Here we extend our previous work in the following directions:

- we provide an overview of the DiSL framework architecture;
- we present the advanced features that contribute to the flexibility and performance of the DiSL framework;
- we present case studies based on reimplementing existing tools using DiSL;
- we provide a tutorial-style introduction to DiSL programming, to help developers of DPA tools to get started with DiSL;
- and finally, we include step-by-step instruction on how to obtain, compile, and run DiSL-based analyses on base programs.

The rest of the paper is structured as follows: Section 2 stems from [24] and introduces the DiSL framework using a simple execution time profiler as a running example, pointing out the advantages of using DiSL to implement the instrumentation. Section 3 provides an overview of the advanced features of DiSL, which also serve as extension points of the DiSL framework. Section 4 presents the high-level architecture of DiSL and provides overview of the instrumentation process. In Section 5 we provide an overview of selected case studies performed during the development of DiSL, and we discuss related work in Section 6. Section 7 concludes the paper.

[1]For performance reasons, DiSL modifies the `java.lang.Thread` class. DiSL also allows an arbitrary user-defined class transformation to become part of the instrumentation process. Both exceptions are discussed in Section 3.5.

2. DiSL by Example

A common example of a dynamic program analysis tool is a method execution time profiler, which usually instruments the method entry and exit join points and introduces storage for timestamps. We describe the main features of DiSL by gradually developing the instrumentation for such a profiler. The same instrumentation is also available on the DiSL home page[2] among the examples. For step-by-step instructions on how to run the examples, please refer to Appendix A.

Note that we intentionally use simple code to illustrate DiSL concepts, oblivious to the overhead it may cause. A developer writing a real instrumentation-based profiler would have to be much more conscious about the overhead introduced by the inserted code. To manage overhead, the developer would have to avoid expensive operations such as memory allocations, string concatenations, or repeated queries for information, as much as possible. To help with that, DiSL provides features that allow caching and passing data between snippets (synthetic local variables and thread local variables), and that allow moving computation of static information to weave time (custom static context). We introduce these features shortly.

2.1. Method Execution Time Profiler

In the first version of our execution time profiler, we simply print the entry and exit times for each method execution as it happens. For that, we need to insert instrumentation at the method entry and method exit join points.

Each DiSL instrumentation is defined through methods declared in standard Java classes. Each method—called *snippet* in DiSL terminology—is annotated so as to specify the join points where the code of the snippet shall be inlined.[3] The profiler instrumentation code on Figure 1 uses two such snippets, the first one prints the entry time, the second one the exit time.

The code uses two annotations to direct inlining. The `@Before` annotation requests the snippet to be inlined before each marked bytecode region (representing a join point); the use of the `@After` annotation places the second snippet after (both normal and abnormal) exit of each marked region. The regions themselves are specified with the `marker` parameter of the annotation. In our example, `BodyMarker` marks the whole method (or constructor) body.

[2]http://disl.ow2.org

[3]The method name can be arbitrarily chosen by the programmer.

```
public class SimpleProfiler {

  @Before(marker=BodyMarker.class)
  static void onMethodEntry() {
    System.out.println("Method entry " + System.nanoTime());
  }

  @After(marker=BodyMarker.class)
  static void onMethodExit() {
    System.out.println("Method exit " + System.nanoTime());
  }
}
```

Figure 1: Instrumenting method entry and exit

The resulting instrumentation thus prints a timestamp upon method entry and exit.

Instead of printing the entry and exit times, we may want to print the elapsed wall-clock time from the method entry to the method exit. The elapsed time can be computed in the after snippet, but to perform the computation, the timestamp of method entry has to be passed from the before snippet to the after snippet.

In traditional AOP languages, which do not support efficient data exchange between advices, this situation would be handled using a local variable within the around advice. In contrast, an instrumentation framework such as DiSL has no need for the usual form of the around advice, which lets the advice code decide whether to skip or proceed with the method invocation [22]. DiSL therefore only supports inlining snippets before and after a particular join point, together with a way for the snippets inlined into the same method to exchange data using *synthetic local variables* [26], as illustrated on Figure 2.

Synthetic local variables are static fields annotated as @SyntheticLocal. The variables have the scope of a method invocation and can be accessed by all snippets that are inlined in the method; that is, they become local variables. Synthetic local variables are initialized to the default value of their declared type (e.g., 0, false, null).

Next, we extend the output of our profiler to include the name of each profiled method. In DiSL, the information about the instrumented class, method, and bytecode region can be obtained through dedicated *static context*

```
public class SimpleProfiler {

  @SyntheticLocal
  static long entryTime;

  @Before(marker=BodyMarker.class)
  static void onMethodEntry() {
    entryTime = System.nanoTime();
  }

  @After(marker=BodyMarker.class)
  static void onMethodExit() {
    System.out.println("Method duration " + (System.nanoTime() - entryTime));
  }
}
```

Figure 2: Passing data between snippets using a synthetic local variable

interfaces. In this case, we are interested in the MethodStaticContext interface, which provides the method name, signature, modifiers and other static data about the intercepted method and its enclosing class. Figure 3 refines the after snippet of Figure 2 to access the fully qualified name of the instrumented method.

```
@After(marker=BodyMarker.class)
static void onMethodExit(MethodStaticContext msc) {
  System.out.println(msc.thisMethodFullName() + " duration "
    + (System.nanoTime() - entryTime));
}
```

Figure 3: Accessing the method name through static context

Static context interfaces provide information that is already available at the instrumentation time. When inlining the snippets, DiSL therefore replaces the calls to these interfaces with the corresponding static context information, thus improving the efficiency of the resulting tools.

DiSL provides a set of static context interfaces, which can be declared as arguments to the snippets in any order. The default set of available interfaces was mainly designed to make DiSL immediately useful to instrumentation developers using AspectJ and ASM. The method static context provides information available in AOP languages such as AspectJ, which we consider to be the minimum. In addition, the set includes interfaces that provide static context information for join points that do not exist in AspectJ, but that we found to be often used in ASM-based DPA tools. This includes basic block static context, which is generally needed in profiling and code coverage analyses with fine-grained resolution, and field access and method invocation static contexts, which are generally needed for shadowing and tracking base program values.

The DiSL programmer may also define custom static context interfaces to perform additional static analysis at instrumentation time or to access information not directly provided by DiSL, but available in the underlying ASM-based bytecode representation.

2.2. Adding Stack Trace

Sometimes knowing the name of the profiled method is not enough. We may also want to know the context in which the method was called. Such context is provided by the stack trace of the profiled method.

There are several ways to obtain the stack trace information in Java, such as calling the getStackTrace() method from java.lang.Thread, but frequent calls to this method may be expensive. Our example therefore obtains the stack trace using instrumentation. Figure 4 shows two additional snippets that maintain the call stack information in a shadow call stack. Upon method entry, the method name is pushed onto the shadow call stack. Upon method exit, the method name is popped off the shadow call stack.

```
@ThreadLocal
static Stack<String> callStack;

@Before(marker=BodyMarker.class, order=1000)
static void pushOnMethodEntry(MethodStaticContext msc) {
  if (callStack == null) { callStack = new Stack<String>(); }
  callStack.push(msc.thisMethodFullName());
}

@After(marker=BodyMarker.class, order=1000)
static void popOnMethodExit() {
  callStack.pop();
}
```

Figure 4: Reifying a thread-specific call stack using dedicated snippets

Each thread maintains a separate shadow call stack, referenced by the thread-local variable callStack.[4] In our example, callStack is initialized for each thread in the before snippet. The thread-local shadow call stack can be accessed from all snippets through the callStack variable; for example, it could be included in the profiler output.

To make sure all snippets observe the shadow call stack in a consistent state, the two snippets that maintain the shadow call stack have to be inserted in a correct order relative to the other snippets. DiSL allows the programmer to specify the order in which snippets matching the same join point should be inlined using the order integer parameter in the snippet annotation. The smaller this number, the closer to the join point the snippet is inlined. In our profiler, the time measurement snippets and the shadow call stack snippets match the same join points (method entry, resp. method exit). We assign a higher order value (1000) to the call stack reification snippets and keep the lower default order value (100) of the snippets for time measurement.[5] Consequently, the callee name is pushed onto the shadow call stack before the entry time is measured, and the exit time is measured before the callee name is popped off the stack.

2.3. Profiling Object Instances

Our next extension addresses situations where the dependency of the method execution time on the identity of the called object instance is of interest. Figure 5 refines the after snippet of Figure 2 by computing the identity hash code of the object instance on which the intercepted method has been called.

```
@After(marker=BodyMarker.class)
static void onMethodExit(MethodStaticContext msc, DynamicContext dc) {
  int identityHC = System.identityHashCode(dc.getThis());
  ...
}
```

Figure 5: Accessing dynamic context information in a snippet

[4]DiSL offers a particularly efficient implementation of thread-local variables with the @ThreadLocal annotation.

[5]If snippet ordering is used, it is recommended to override the value in all snippets for improved readability.

The snippet uses the DynamicContext *dynamic context interface* to get a reference to the current object instance. Similar to the static context interfaces, the dynamic context interfaces are also exposed to the snippets as method arguments. Unlike the static context information, which is resolved at instrumentation time, calls to the dynamic context interface are replaced with code that obtains the required dynamic information at runtime. Besides the object reference used in the example, DiSL provides access to other dynamic context information including the local variables, the method arguments, and the values on the operand stack.

2.4. Selecting Profiled Methods

Often, it is useful to restrict the instrumentation to certain methods. For example, we may want to profile only the execution of methods that contain loops, because such methods are likely to contribute more to the overall execution time.

DiSL allows programmers to restrict the instrumentation scope using the *guard* construct. A guard is a user-defined class whose one method carries the @GuardMethod annotation. This method determines whether a snippet matching a particular join point is inlined. Figure 6 shows the signature of a guard restricting the instrumentation only to methods containing loops. The body of the methodContainsLoop() guard method, not shown here, would implement the detection of a loop in a method. A loop detector based on control flow analysis is included as part of DiSL.

```
public class MethodsContainingLoop {

  @GuardMethod
  public static boolean methodContainsLoop() {
    ... // Loop detection based on control flow analysis
  }
}
```

Figure 6: Skeleton of a guard for selecting only methods containing a loop

The loop guard is associated with a snippet using the guard annotation parameter, as illustrated in Figure 7. Note that the loop guard is not used in the shadow call stack snippets. We want to maintain complete stack trace information without omitting the methods that do not contain loops.

```
@Before(marker=BodyMarker.class, guard=MethodsContainingLoop.class)
static void onMethodEntry() { ... }

@After(marker=BodyMarker.class, guard=MethodsContainingLoop.class)
static void onMethodExit(...) { ... }
```

Figure 7: Applying time measurement snippets only in methods containing a loop

3. Advanced DiSL Features

The features presented so far cover basic DiSL usage. We continue with examples illustrating the more advanced features of DiSL. These can be roughly split into two categories.

The first category includes method argument processing, using other markers from the DiSL marker library, and creating custom static context implementations. These features mostly require the developer to be familiar with DiSL, and some may need a basic knowledge of Java bytecode. The requirements for using a custom static context depend on the actual usage. For example, generating custom names for code locations using the information already available in other static contexts just requires the developer to use the existing API, and to adhere to the requirements for static context implementations.

The second category again includes custom static context, as well as custom custom markers and bytecode transformers. Compared to the previous case, here we consider using custom static context for more complex tasks, such as performing a custom static analysis. Custom markers allow definition of new join points, and custom transformer are just hooks into the instrumentation process, where anything is allowed. Using these features basically means extending DiSL, which will require the developer to have a solid knowledge of DiSL, ASM, and Java bytecode.

The need for features from the first category will come gradually, as a result of using DiSL for more sophisticated instrumentations. The features in the second category are really meant for extending DiSL, and we anticipate that most DiSL users will never use them. We now review each of the features in turn, in the order of increasing developer requirements.

3.1. Analyzing Method Arguments

DiSL provides two different mechanisms for analyzing method arguments. The first approach provides the method arguments to the snippet in an object array. The entire array is constructed dynamically at runtime, with arguments of primitive types boxed. Conceptually simple, the approach requires object allocation and always processes all arguments.

The second approach aims at situations where the overhead of using object arrays is not acceptable. The approach uses code fragments called *argument processors*. Each argument processor analyzes only one type of method arguments. The code of the argument processor is inlined into the snippet where it is applied. With argument processors, it is possible to access method arguments without object allocation.

Technically, the argument processor is an annotated Java class containing argument processing methods. The first argument of each argument processor method is of the type being processed, that is, any basic Java type (int, byte, double ...), String, or an object reference. As additional arguments, the methods can receive dynamic or static contexts, including *argument context*, which is a special kind of static context available only within the argument processor. The ArgumentContext interface exposes information about the currently processed argument and can be used to limit argument processing only to arguments at a particular position or with a particular type. The argument processor methods can also use thread-local or synthetic local variables.

An example of an argument processor that processes int arguments is given in Figure 8.

```
@ArgumentProcessor
public class IntArgumentPrinter {
  public static void printIntegerArgument (
    int val, ArgumentContext ac, MethodStaticContext msc) {

    System.out.printf(
      "Int argument value in method %s at position %d of %d is %d\n",
      msc.thisMethodFullName(), ac.getPosition(), ac.getTotalCount(), val
    );
  }
}
```

Figure 8: A simple argument processor for printing the values of integer arguments

The argument processor is used by applying it in an argument processor context within a snippet. The argument processor context can apply an argument processor in two modes. All snippets can apply the processor on the arguments of the current method. Snippets inserted just before a method invocation can also apply the processor on the invocation arguments. Figure 9 shows a snippet that uses the IntArgumentPrinter argument processor from Figure 8 to print out the values of the integer arguments of the currently executed method.

```
@Before(marker = BodyMarker.class)
public static void onMethodEntry(ArgumentProcessorContext apc) {
  apc.apply(IntArgumentPrinter.class, ArgumentProcessorMode.METHOD_ARGS);
}
```

Figure 9: Using an argument processor within a snippet

3.2. Join Point Marker Library

In all the examples presented earlier, profiles were collected with method granularity. Such profiles may be insufficient when profiling long methods with loops and nested invocations. In these cases, a more fine grained measurement can help identify the problematic parts of the long methods.

In the profiler example, a more fine grained measurement can be achieved using a different marker with the profiling snippets. DiSL provides a library of markers (e.g., BasicBlockMarker, BytecodeMarker) for intercepting many common bytecode patterns; Figure 10 illustrates the use of BasicBlockMarker for basic block profiling.

As presented, the change only impacts the choice of the marker class. Although the resulting instrumentation is valid, the resulting profile is of limited use because it lacks the identification of the basic blocks being profiled. We add this identification next.

3.3. Custom Static Context

There are multiple options for identifying a basic block in the profiler example. We can use the ordinal number of the basic block as made available by the BasicBlockStaticContext; however, such identification is only useful if the information about the correspondence between the basic block numbers and the profiled code is available when interpreting the results. The source code line number is a valuable alternative when working at the source code level, however, the identification is not necessarily unique and the need for additional information when interpreting the results also persists. To provide

```
@Before(marker=BasicBlockMarker.class)
static void onBasicBlockEntry() { ... }

@After(marker=BasicBlockMarker.class)
static void onBasicBlockExit(...) { ... }
```

Figure 10: Writing snippets to profile entry and exit from basic blocks

an example of custom static context, we illustrate a third option, namely identifying the basic block by the ordinal number of its first instruction and its length, counted in the number of instructions (numbers are valid for uninstrumented code). Implementing the other two approaches in DiSL is of similar complexity.

The identification of a basic block is conceptually a part of the static context of each snippet, and it would ideally be available through one of the existing static context interfaces. In this particular case, DiSL actually provides such an interface. However, this may not be true in general. While we aim to equip DiSL with a rich library of static context interfaces offering all the information that may be required by an analysis tool, we can only guess at what information will other analyses—especially new ones—require.

In the two years of development and evaluation, we found the set of static context interfaces provided by DiSL to be sufficient for almost all DPA tools we recasted or developed. A notable exception was an analysis that required static context information for loops, which required performing dominator analysis on basic blocks at weave time. Other than that, most analyses used tiny custom static context implementations that were difficult to generalize—either to obtain information related to a particular type of bytecode instructions (which is easily available from the underlying ASM-based code representation), or to precompute trace messages with static information at weave time (to avoid string concatenation at runtime).

Consequently, DiSL contains mostly static context implementations providing information that was generally useful in the tools that we have recasted so far, including a few that were used rarely, but had non-trivial implementation, such as the loop static context. Since we cannot anticipate what static information will be required by all analyses, we allow DiSL users to define a custom static context. However, based on our experience, we expect most users to use a custom static context mainly for generating names at weave time, which allows embedding these values in the snippet code as constants.

```
public class BasicBlockID extends AbstractStaticContext {
  public String getID() {
    // validate that the basic block has only one end
    ...

    // get starting and ending instruction from marker
    AbstractInsnNode startInsn = staticContextData.getRegionStart();
    AbstractInsnNode endInsn = staticContextData.getRegionEnds().get(0);

    // traverse entire method code and calculate instruction index
    int bbStart = -1;
    int bbLength = 0;
    boolean startFound = false;
    boolean endFound = false;
    InsnList code = staticContextData.getMethodNode().instructions;
    for(AbstractInsnNode insn = code.getFirst();
        insn != null; insn = insn.getNext()) {

      // increase block start index until start instruction found
      if(!startFound) {
        if(insn.getOpcode() != -1) ++bbStart;
        startFound = (insn == startInsn);
      }

      if(startFound) {
        // count instructions and exit when end instruction found
        if(insn.getOpcode() != -1) ++bbLength;
        if(insn == endInsn) {
          endFound = true;
          break;
        }
      }
    }

    // validate that both start and end were found
    ...

    // construct and return the basic block ID
    return bbStart + "(" + bbLength + ")";
  }
}
```

Figure 11: Custom static context computing a basic block ID

Figure 11 illustrates a custom static context that serves as the basic block ID calculator. A custom static context is a standard Java class that extends the AbstractStaticContext class or implements the StaticContext interface directly. The methods of the custom static context class have no arguments and return a basic type or String. The BasicBlockID class from Figure 11 contains one such method, getID(), which computes the ID of a basic block.

The computation queries the first and the last instruction of the region identified by the basic block marker. After that, it iterates over the code of the entire method, first incrementing the block index until the basic block start is reached, then incrementing the block length until the basic block end is found. The method returns the ID as String whose first part is the index and second part the length.

Custom static context methods can access the current static context information through a protected field called staticContextData. The available information describes the marked region, snippet, method, and class where the custom static context is used. The region description includes one starting instruction and one or more ending instructions depending on the marker. The snippet structure holds all the information connected to the snippet where the static context is used. The method and class data are represented by ASM objects MethodNode and ClassNode.

3.4. Custom Bytecode Marker

It is not always possible to profile a method by instrumenting its body. For example, the method can be implemented in native code or can execute remotely. To profile such methods, the instrumentation has to be placed around the method invocation.

In DiSL, method invocation can be easily captured by the BytecodeMarker with adequate parameters. To illustrate the extensibility of DiSL, we instead implement a new custom marker that captures method invocations, displayed in Figure 12.

The role of a marker is to select the bytecode regions for instrumentation. A custom bytecode marker in DiSL must implement the Marker interface. Typically, the marker would not implement this interface directly, but instead inherit from the AbstractDWRMarker abstract class, which also takes care of correctly placing the weaving points. In our example, the MethodInvocationMarker class traverses all instructions using ASM and creates a single-instruction region for each method invocation encountered; the

```
public class MethodInvocationMarker extends AbstractDWRMarker {
  public List<MarkedRegion> markWithDefaultWeavingReg(MethodNode method) {

    List<MarkedRegion> regions = new LinkedList<MarkedRegion>();

    // traverse all instructions
    InsnList instructions = method.instructions;
    for (AbstractInsnNode instruction : instructions.toArray()) {

      // check for method invocation instructions
      if (instruction instanceof MethodInsnNode) {

        // add region containing one instruction (method invocation)
        regions.add(new MarkedRegion(instruction, instruction));
      }
    }

    return regions;
  }
}
```

Figure 12: Custom marker implementing a method invocation join point

abstract marker class is used to compute all the weaving information automatically.

Note that the example marker captures all method invocations. To reduce the instrumentation scope, the developer should use either a guard or a runtime check.

3.5. Custom Bytecode Transformer

DiSL is designed for writing tools that observe the application without modifying its behavior. It will refuse to insert snippets that would change the application control flow, modify fields, or insert methods or fields to classes. The only exception is the modification of the java.lang.Thread class performed by DiSL to provide very efficient implementation of thread-local variables.[6]

[6]DiSL adds a field to the java.lang.Thread class for every thread-local variable, which significantly outperforms an approach based on the java.lang.ThreadLocal class. Here, the performance benefits far outweigh the chance of breaking application behavior, because the modifications are limited to a single system class.

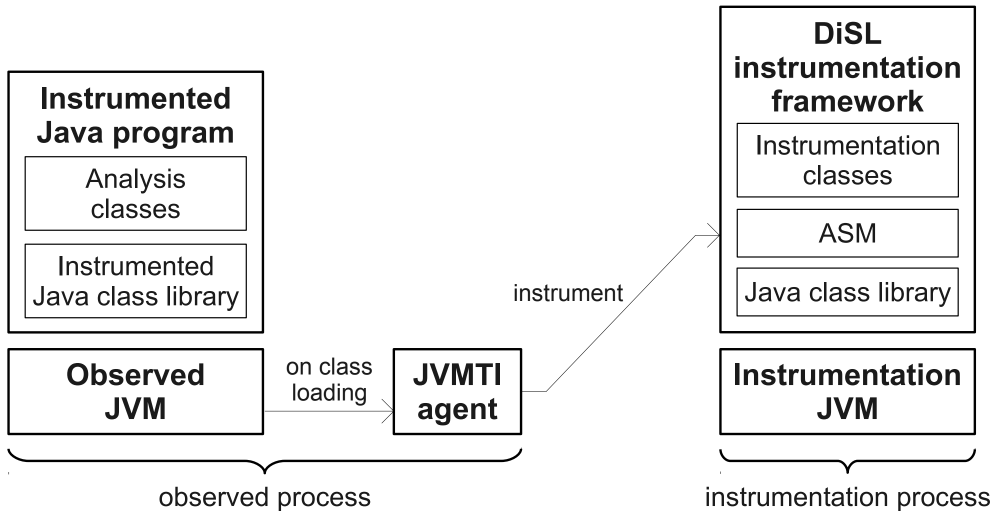

Figure 13: Architecture of DiSL.

However, in special cases, a tool implementation may require application modifications beyond what DiSL allows. This may result if a tool needs to perform structural modifications to the application code, or when the method-level scope provided by DiSL is too narrow. In these cases, DiSL can invoke a custom transformer to modify the class just before it is instrumented.

Custom transformers have to implement the `ch.usi.dag.disl.Transformer` interface to receive raw class data from DiSL, and to return the modified data back to DiSL. The class data is passed around as an array of bytes, and apart from the `Transformer` interface, DiSL neither provides any API for class transformation, nor mandates the use of any particular bytecode manipulation framework. The developer is free to modify the class data in any way, typically with the help of a bytecode manipulation framework (e.g., ASM), that is able to parse class from an array of bytes and return the result in the same form.

4. DiSL Architecture and Instrumentation Process

To minimize perturbation in the observed program, DiSL performs bytecode instrumentation within a separate Java Virtual Machine (JVM) process, that is, the *instrumentation process*. In this way, class loading and initialization triggered by the instrumentation framework do not happen within the *observed process*.

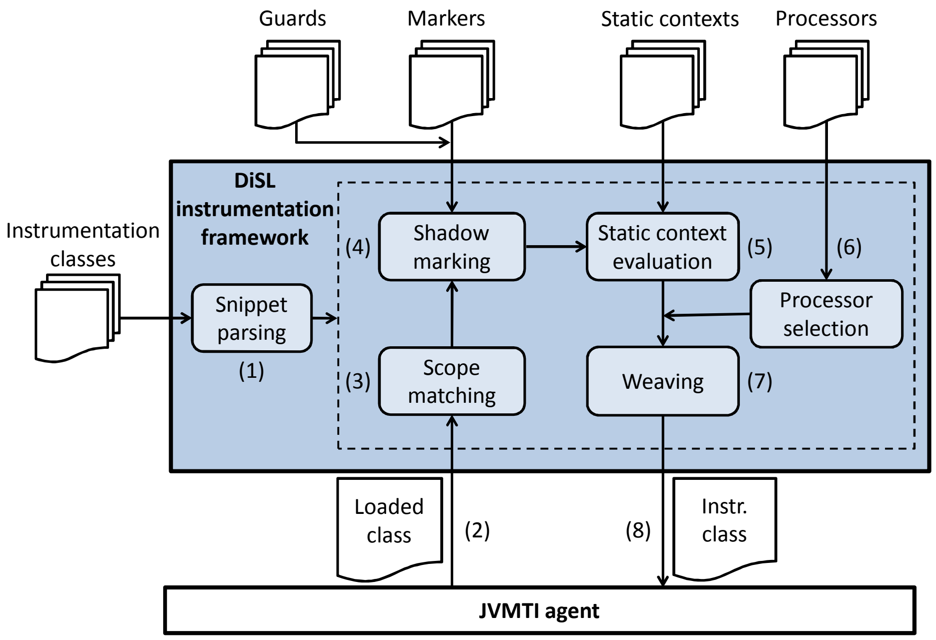

Figure 14: Overview of the DiSL instrumentation process.

As illustrated in Figure 13, a native JVMTI agent[7] captures all class loading events (starting with java.lang.Object) in the observed JVM and sends every class as a byte array to the DiSL instrumentation framework through a socket. Here, DiSL uses ASM for instrumentation and relies on *polymorphic bytecode instrumentation* [25] to ensure complete bytecode coverage. All classes are instrumented only once, whenever they are loaded by the JVM. While dynamic instrumentation and reinstrumentation at runtime is a work in progress, it is currently not supported by DiSL.

Figure 14 gives an overview of the DiSL instrumentation process. 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 the JVMTI agent (step 2), the instrumentation process starts with the snippet selection. The selection is done in three phases, starting with scope matching (step 3). Then, bytecode regions are marked using the markers associated with the snippets selected in the previous phase. Finally, marked bytecode

[7]http://download.oracle.com/javase/6/docs/platform/jvmti/jvmti.html

regions are evaluated by guards and only snippets with at least one valid marked region are selected (step 4).

At this point, all snippets that will be used for instrumentation are known. Static contexts are used to compute the static information required by snippets (step 5). As described in Section 3, custom static contexts allow programmers to declare expressions to be evaluated at instrumentation time. However, if such expressions do not require custom context information, they can simply be embedded in the snippet code. In fact, DiSL can be configured to perform partial evaluation of inlined snippets [23]. This optimization can simplify certain snippet constructs, for example it can make conditional branching on static information in the snippet as efficient as using a guard.

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 instrumentation (step 7). Argument processors are applied, and calls to static contexts are replaced with the computed static information. The framework also generates the bytecodes to access dynamic context information. To prevent the instrumentation code from throwing exceptions that could modify the control flow in the observed program, DiSL automatically inserts code intercepting all exceptions originating from the snippets, reporting an error if an exception is thrown (and not handled) by the instrumentation. Finally, the instrumented class is returned to the observed JVM (step 8) where it is linked. For a more detailed description of the process, we refer the interested reader to [22].

5. Case Studies

To demonstrate the benefits of DiSL when developing instrumentation for DPA tools, we present several case studies involving existing DPA tools. The instrumentation parts of those tools were originally implemented either using AspectJ, or the ASM bytecode manipulation library. For evaluation purposes, we reimplemented the instrumentation part of each tool using DiSL and compared both the conciseness and the performance of the DiSL-based instrumentation with the original implementation. We first give a short overview of the recasted tools, and then proceed with the evaluation.

5.1. Tool Descriptions

JP2 [27] is a calling-context profiler for languages targeting the JVM. JP2 uses ASM-based instrumentation to collect various static (i.e., method names, number and sizes of basic blocks) and dynamic metrics (i.e., method invocations, basic block executions, and number of executed bytecodes). For each method, the metrics are associated with a corresponding node in a calling-context tree (CCT) [28], grouped by the position of the method call-site in the caller. The collected information can be then used for both inter- and intra-procedural analysis of the application.

JCarder[8] is a tool for finding potential deadlocks in multi-threaded Java applications. JCarder uses ASM-based instrumentation to construct a dependency graph for threads and locks at runtime, and if the graph contains a cycle, JCarder reports a potential deadlock.

Senseo [8] is a tool for profiling and code comprehension. For each method invocation, Senseo collects calling-context specific information, which a plugin[9] then makes available to the user via enriched code views in the Eclipse IDE. Senseo uses AspectJ-based instrumentation to count method invocations, object allocations, and to collect statistics on method arguments and return types.

RacerAJ [20] is a tool for finding potential data races in multi-threaded Java applications. RacerAJ uses AOP-based instrumentation to monitor all field accesses and lock acquisitions/releases, and reports a potential data race when a field is accessed from multiple threads without holding a lock that synchronizes the accesses.

Besides recasting existing tools for evaluation purposes, we successfully used DiSL to develop new field immutability and field sharing analyses, which have been used to compare Java and Scala workloads [29]. These analyses are now part of a comprehensive toolchain for workload characterization for Java and other languages targeting the JVM [30].

5.2. Evaluating Instrumentation Conciseness

In our experience with developing DPA tools with AspectJ and ASM, we consistently found instrumentations developed using the AOP pointcut/advice model very clear and concise—provided that an instrumentation could be expressed using the available join points. On the other hand, using ASM allowed us to perform basically any kind of instrumentation with significantly better performance, at the cost of very low-level and verbose instrumentation

[8]http://www.jcarder.org/
[9]http://scg.unibe.ch/research/senseo

	JP2		**JCarder**		**Senseo**		**RacerAJ**	
	DiSL	ASM	DiSL	ASM	DiSL	AspectJ	DiSL	AspectJ
Physical LOC	96	477	89	650	74	44	136	33
Logical LOC	64	375	64	399	44	19	78	24

Table 1: The amount of code (in source lines of code) comprising the instrumentation parts of the analysis tools, written using DiSL, ASM, and AspectJ.

code, which was rather fragile and thus difficult to maintain. With DiSL, we strive for the simplicity and conciseness of the AOP-based instrumentations, without sacrificing expressiveness and performance provided by ASM.

To evaluate how DiSL compares to AspectJ and ASM in terms of instrumentation conciseness, we consider the amount of code—measured in source lines of code (SLOC)—that needs to be written in AspectJ, ASM, and DiSL to implement an equivalent instrumentation for the evaluated tools. While there is no proof that "less code" automatically translates to "better code", we believe that in this particular context, an implementation that is significantly shorter (i.e., more concise) due to use of better fitting abstractions can be considered easier to write, understand, and maintain. This view is also supported by the results of a controlled user study [31], where DiSL was shown to reduce instrumentation development time and improve instrumentation correctness compared to ASM.

Table 1 summarizes the SLOC counts[10] of both the DiSL-based and the original implementations of instrumentation for each tool. The number of physical SLOC illustrates the overall size of the implementation, while the number of logical SLOC captures the amount of code essential for the implementation. In our comparison, we use the logical SLOC count as an indicator of conciseness for implementations of equivalent instrumentations.

We observe that for the evaluated DPA tools, the DiSL-based implementations of their instrumentation parts require less code than their ASM-based equivalents. This is because because bytecode manipulation, even when using ASM, results in very verbose code. On the other hand, the DiSL-based instrumentations require more code than their AOP-based equivalents. This can be partially attributed to DiSL being an embedded language hosted in

[10]Calculated using Unified CodeCount by CSSE USC, rel. 2011.10, http://sunset.usc.edu/research/CODECOUNT.

Benchmark	Description
avrora	Simulates a number of programs run on a grid of AVR microcontrollers.
batik	Produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik.
eclipse	Executes some of the (non-gui) JDT performance tests for the Eclipse IDE.
fop	Takes an XSL-FO file, parses it, formats it, and generates a PDF file.
h2	Executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application.
jython	Inteprets a the pybench Python benchmark.
luindex	Uses lucene to index a set of documents comprising the works of Shakespeare and the King James Bible.
lusearch	Uses lucene to do a keyword search over a corpus of data comprising the works of Shakespeare and the King James Bible.
pmd	Analyzes a set of Java classes for a range of source code problems.
sunflow	Renders a set of images using ray tracing.
xalan	Transforms XML documents into HTML.
tomcat	Runs a set of queries against a Tomcat server retrieving and verifying the resulting web pages.
tradebeans	Runs the `daytrader` benchmark via Java Beans to a GERONIMO back-end with an in-memory `h2` as the underlying database.
tradesoap	Runs the `daytrader` benchmark via SOAP to a GERONIMO back-end with an in-memory `h2` as the underlying database.

Table 2: Overview of benchmarks from the DaCapo suite. Benchmarks at the bottom were excluded from evaluation due to hard-coded timeouts and well-known problems.

Java, whereas AOP has the advantage of a separate language. Moreover, DiSL instrumentations also include code that is evaluated at instrumentation time, which increases the source code size, but provides significant performance benefits at runtime [22]. However, in the context of instrumentations for DPA, DiSL is more flexible and expressive than AOP without impairing the performance of the resulting tools, as shown in the following section.

5.3. Evaluating Instrumentation Performance

To evaluate the instrumentation performance, we compare the execution time of the original and the recasted tools on benchmarks from the DaCapo [32] suite (release 9.12). Table 2 provides an overview of the DaCapo benchmarks,

	JP2		JCarder		Senseo		RacerAJ	
	ASM	DiSL	ASM	DiSL	AspectJ	DiSL	AspectJ	DiSL
avrora	9.73	10	1.31	1.83	4.48	2.68	110.29	32.58
batik	4.77	5.5	1.05	1.05	2.18	1.43	31.23	6.64
eclipse	3.08	3.46	1.01	1.53	8.28	5.97	11.94	14.82
fop	6.17	7.94	1.1	1.32	9.9	4.54	49.06	12.56
h2	12.21	13.8	2.11	3.79	10.64	4.14	157.7	79.07
jython	26.28	30.36	7.92	4.54	2.19	1.52	236.8	104.4
luindex	2.63	2.88	1.25	1.26	9.34	3.06	60.31	16.25
lusearch	24.18	31.82	7.78	8.09	7.88	3.7	147.71	49.06
pmd	4.8	7.4	1.07	1.07	3.52	2.09	53.8	24
sunflow	10.53	9.85	1.05	1.03	39.28	12.52	398.18	211.22
xalan	27.94	26.22	26.08	28.19	29.25	5.05	68.89	29.33
geomean	8.83	10.09	2.24	2.47	7.69	3.5	81.01	32.26

Table 3: Overhead factors for original and recasted tools when run on the benchmarks from the DaCapo suite.

as presented on the DaCapo suite web site.[11] Of the fourteen benchmarks present in the suite, we excluded the tradeswap, tradebeans, and tomcat due to well known issues[12] unrelated to DiSL. All experiments were run on a multicore platform[13] with all non-essential system services disabled.

We report results for steady-state performance in Table 3. For each benchmark, we report a steady-state overhead factor, determined from a single run with 10 iterations of each benchmark, with the first 5 iterations excluded to minimize the influence of startup transients and interpreted code. The number of iterations to exclude was determined by visual inspection of the data from the benchmarks. As a summary metric, we also report the geometric mean of overhead factors from all benchmarks.

The results in Table 3 indicate that the recasted tools are typically roughly as fast as their ASM-based counterparts (JP2, JCarder), but never much slower. Significant performance improvements can be observed in the case of

[11] http://www.dacapobench.org/benchmarks.html

[12] See bug ID 2955469 (hardcoded timeout in tradesoap and tradebeans) and bug ID 2934521 (StackOverflowError in tomcat) in the DaCapo bug tracker at http://sourceforge.net/tracker/?group_id=172498&atid=861957.

[13] Four quad-core Intel Xeon CPUs E7340, 2.4 GHz, 16 GB RAM, Ubuntu GNU/Linux 11.04 64-bit with kernel 2.6.38, Oracle Java HotSpot 64-bit Server VM 1.6.0_29.

AOP-based tools (Senseo, RacerAJ), which can be attributed mainly to the fact that DiSL allows to use static information at instrumentation time to precisely control where to insert snippet code, hence avoiding costly checks and static information computation (often comprising string concatenations) at runtime. Additional performance gains can be attributed to the ability of DiSL snippets to efficiently access the constant pool and the JVM operand stack, which is particularly relevant in comparisons with AOP-based tools.

6. Related Work

In previous work, we presented @J [21], a Java annotation-based AOP language for simplifying dynamic analysis. Compared to DiSL, @J does not provide an open join point model and efficient access to dynamic context information. DiSL guards can be emulated by means of staged advice, where weave-time evaluation of advice yields runtime residues that are woven. However, this requires the use of synthetic local variables and a more complex composition of snippets.

In [33] 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 can 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 [34] or Chord[14] ease composition of a set of dynamic analyses. However, such approaches do not support an open join point model and the set of context information that can be accessed at intercepted code regions is not extensible.

In [35], a meta-aspect protocol (MAP) is proposed to separate the host language from analysis-specific aspect languages. MAP supports an open join point model and advanced deployment methods (i.e., global, per object, and per block). While MAP allows fast prototyping of custom analysis languages, it does not focus on high efficiency of the developed analysis tools.

Josh [36] 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 that are readily available in DiSL.

[14]http://pag.gatech.edu/chord/

The approach described in [37] enables customized pointcuts that are partially evaluated at weave-time, using a declarative language to define the bytecode regions to be marked. Because only a subset of bytecodes is converted to the declarative language, it is not possible to define basic block pointcuts as in DiSL.

Prevailing AspectJ weavers lack support for embedding custom static analysis in the weaving process. In [38] compile-time statically executable advice is proposed, which is similar to static context in DiSL. SCoPE [39] 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, allow weave-time conditional evaluation and prevent the insertion of dead code.

The AspectBench Compiler (*abc*) [40] eases the implementation of AspectJ extensions. As intermediate representation, *abc* uses Jimple to mark bytecode regions. Jimple has no information on 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.

Javassist [17] is a load-time bytecode manipulation library that allows load-time structural reflection and definition of new classes at runtime. The API provides 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.

Shrike [16] is a bytecode instrumentation library that is part of the T.J. Watson Libraries for Analysis (WALA) [41] and provides interesting features to increase efficiency. For example, parsing is limited to the parts of the class to be modified, bytecode instructions are represented by immutable objects, and many constant instructions can be represented with a single object shared between methods. Moreover, Shrike has a patch-based API that atomically applies all modifications to a given method body and automatically splits up methods larger than the limit imposed by the Java class file format. Dila [42] is another library of WALA that relies on Shrike for load-time bytecode modifications.

Compared to DiSL, Javassist and Shrike do not follow a pointcut/advice model and do not provide built-in support for basic-block analysis and synthetic local variables.

7. Conclusion

This paper is the first complete presentation of DiSL, a domain-specific language and framework designed specifically for instrumentation-based dynamic program analysis. DiSL occupies a unique position among the existing instrumentation frameworks:

- DiSL instrumentations are concise—rather than relying on low-level bytecode manipulation constructs, DiSL adopts a high-level pointcut/advice model inspired by AOP, which leads to compact and readable code.
- DiSL instrumentations are flexible—the DiSL framework provides an open join-point model that allows instrumenting any bytecode sequence, coupled with techniques that extend the instrumentation coverage to any method with bytecode representation.
- DiSL instrumentations are efficient—static context information is precomputed and embedded in the inserted code as constants; dynamic context information can be accessed efficiently through special methods.

DiSL is built on top of ASM [13], the well-known bytecode manipulation library, and is itself an open-source project.[15]

Various isolated aspects of the DiSL framework were presented in previous work [22, 23, 24]. This paper extends our previous presentation of DiSL—we provide an overview of the DiSL architecture, introduce the DiSL programming model and illustrate the basic concepts with a running example. For advanced developers, we demonstrate the use of two extension points of the DiSL framework—custom static contexts and custom bytecode markers. These extension points allow the developers to introduce new types of static information that can be used within snippets and guards, and to extend the set of join points recognized by DiSL. Finally, we present several case studies that demonstrate successful deployment of DiSL-based DPA tools.

We believe that the unique combination of the high-level programming model with the flexibility and detailed control of low-level bytecode instrumentation makes DiSL a valuable tool that can reduce the effort needed for developing new dynamic analysis tools and similar software applications running in the JVM. Since the implementation of DiSL is specific to JVM, it

[15]The project is hosted by the OW2 consortium at http://disl.ow2.org

is mostly the programming language and software engineering communities targeting the JVM—both in academia and in industry—who can benefit most from DiSL. However, none of the concepts employed in DiSL are JVM specific, which makes it possible, given resources, to implement a similar tool for other managed platforms, such as the Common Language Runtime.

Acknowledgments

The research presented in this paper has been supported by the Swiss National Science Foundation (project CRSII2_136225), by the European Commission (Seventh Framework Programme projects 287746 and 257414), by the Czech Science Foundation (project GACR P202/10/J042), and by the Sino-Swiss Science and Technology Cooperation (SSSTC) Institutional Partnership (project IP04–092010).

References

[1] M. Jovic, A. Adamoli, M. Hauswirth, Catch me if you can: performance bug detection in the wild, in: Proceedings of the 2011 ACM international conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'11), ACM, 2011, pp. 155–170.

[2] W. Binder, J. Hulaas, P. Moret, A. Villazón, Platform-independent profiling in a virtual execution environment, Software: Practice and Experience 39 (2009) 47–79.

[3] M. Eaddy, A. Aho, W. Hu, P. McDonald, J. Burger, Debugging aspect-enabled programs, in: Proceedings of the 6th international conference on Software Composition (SC'07), Springer, 2007, pp. 200–215.

[4] S. Artzi, S. Kim, M. D. Ernst, ReCrash: Making software failures reproducible by preserving object states, in: Proceedings of the 22th European Conference on Object-Oriented Programming (ECOOP'08), volume 5142 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 542–565.

[5] G. Xu, A. Rountev, Precise memory leak detection for Java software using container profiling, in: Proceedings of the 30th International Conference on Software engineering (ICSE'08), ACM, 2008, pp. 151–160.

[6] F. Chen, T. F. Serbanuta, G. Rosu, jPredictor: A predictive runtime analysis tool for Java, in: Proceedings of the 30th International Conference on Software Engineering (ICSE'08), ACM, 2008, pp. 221–230.

[7] NetBeans, The NetBeans Profiler Project, Web pages at http://profiler.netbeans.org/, 2012.

[8] D. Röthlisberger, M. Härry, W. Binder, P. Moret, D. Ansaloni, A. Villazon, O. Nierstrasz, Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks, IEEE Transactions on Software Engineering 38 (2012) 579–591.

[9] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni, Model-based performance prediction in software development: a survey, IEEE Transactions on Software Engineering 30 (2004) 295–310.

[10] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, G. Kappel, A Survey on Aspect-Oriented Modeling Approaches, Technical Report, Vienna University of Technology, Johannes Kepler University Linz, 2007.

[11] R. France, I. Ray, G. Georg, S. Ghosh, Aspect-oriented approach to early design modelling, IEE Proceedings – Software 151 (2004) 173–185.

[12] D. Simmonds, A. Solberg, R. Reddy, R. France, S. Ghosh, An aspect oriented model driven framework, in: Enterprise Computing Conference (EDOC), pp. 119–130.

[13] OW2 Consortium, ASM – A Java bytecode engineering library, Web pages at http://asm.ow2.org/, 2012.

[14] The Apache Jakarta Project, The Byte Code Engineering Library (BCEL), Web pages at http://jakarta.apache.org/bcel/, 2012.

[15] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, V. Sundaresan, Optimizing Java bytecode using the Soot framework: Is it feasible?, in: Proceedings of the 9th international conference on Compiler Construction (CC'00), volume 1781 of *Lecture Notes in Computer Science*, Springer, 2000, pp. 18–34.

[16] IBM, Shrike Bytecode Instrumentation Library, Web pages at http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview, 2012.

[17] 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*, Springer, 2000, pp. 313–336.

[18] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), volume 1241 of *Lecture Notes in Computer Science*, Springer, 1997, pp. 220–242.

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

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

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

[22] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, Z. Qi, DiSL: a domain-specific language for bytecode instrumentation, in: Proceedings of the 11th international conference on Aspect-oriented Software Development (AOSD'12), ACM, 2012, pp. 239–250.

[23] Y. Zheng, D. Ansaloni, L. Marek, A. Sewe, W. Binder, A. Villazón, P. Tuma, Z. Qi, M. Mezini, Turbo DiSL: Partial evaluation for high-level bytecode instrumentation, in: Objects, Models, Components, Patterns, volume 7304 of *Lecture Notes in Computer Science*, pp. 353–368.

[24] L. Marek, Y. Zheng, D. Ansaloni, A. Sarimbekov, W. Binder, P. Tůma, Z. Qi, Java bytecode instrumentation made easy: The DiSL framework for dynamic program analysis, in: Proceedings of the 10th Asian Symposium on Programming Languages and Systems (APLAS'12), volume 7705 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 256–263.

[25] P. Moret, W. Binder, É. Tanter, Polymorphic bytecode instrumentation, in: Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD'11), ACM, 2011, pp. 129–140.

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

[27] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, M. Mezini, JP2: Call-site aware calling context profiling for the Java Virtual Machine, Science of Computer Programming (2012).

[28] G. Ammons, T. Ball, J. R. Larus, Exploiting hardware performance counters with flow and context sensitive profiling, in: Proceedings of the ACM SIGPLAN 1997 conference on Programming Language Design and Implementation (PLDI'97), ACM, 1997, pp. 85–96.

[29] A. Sewe, M. Mezini, A. Sarimbekov, D. Ansaloni, W. Binder, N. Ricci, S. Z. Guyer, new Scala() instanceof Java: A comparison of the memory behaviour of Java and Scala programs, in: Proceedings of the International Symposium on Memory Management (ISMM'12), ACM, 2012, pp. 97–108.

[30] A. Sarimbekov, A. Sewe, S. Kell, Y. Zheng, W. Binder, L. Bulej, D. Ansaloni, A comprehensive toolchain for workload characterization across JVM languages, in: Proceedings of the 11th ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE'13), ACM, 2013, pp. 9–16.

[31] A. Sarimbekov, Y. Zheng, D. Ansaloni, L. Bulej, L. Marek, W. Binder, P. Tůma, Z. Qi, Productive development of dynamic program analysis tools with DiSL, in: Proceedings of the 22nd Australasian Software Engineering Conference (ASWEC'13), IEEE, 2013, pp. 11–19.

[32] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The DaCapo benchmarks: Java benchmarking development and analysis, in: Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented

Programing, Systems, Languages, and Applications (OOPSLA'06), ACM, 2006, pp. 169–190.

[33] W. Binder, P. Moret, D. Ansaloni, A. Sarimbekov, A. Yokokawa, E. Tanter, Towards a domain-specific aspect language for dynamic program analysis: position paper, in: Proceedings of the 6th workshop on Domain-specific Aspect Languages (DSAL'11), ACM, 2011, pp. 9–11.

[34] C. Flanagan, 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), ACM, 2010, pp. 1–8.

[35] M. Achenbach, K. Ostermann, A meta-aspect protocol for developing dynamic analyses, in: Proceedings of the 1st international conference on Runtime Verification (RV'10), volume 6418 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 153–167.

[36] S. Chiba, K. Nakagawa, Josh: An open AspectJ-like language, in: Proceedings of the 3rd international conference on Aspect-Oriented Software Development (AOSD'04), ACM, 2004, pp. 102–111.

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

[38] K. Lieberherr, D. H. Lorenz, 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), ACM, 2003, pp. 40–49.

[39] T. Aotani, H. Masuhara, SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts, in: Proceedings of the 6th international conference on Aspect-oriented Software Development (AOSD'07), ACM, 2007, pp. 161–172.

[40] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, abc: An extensible AspectJ compiler, in: Proceedings of the 4th international conference on Aspect-Oriented Software Development (AOSD'05), ACM, 2005, pp. 87–98.

[41] IBM, Watson Libraries for Analysis (WALA), Web pages at http://wala.sourceforge.net/wiki/index.php/Main_Page, 2012.

[42] IBM, Dynamic Load-time Instrumentation Library for Java (Dila), Web pages at http://wala.sourceforge.net/wiki/index.php/GettingStarted:wala.dila, 2012.

Appendix A. Running DiSL

This appendix provides a step-by-step guide to download, compile, and run the DiSL framework. The current release of DiSL is tested with Java 6 and Java 7 on Linux, for which we provide scripts to compile and run the framework. Compatibility with other platforms will be added in future releases. To build DiSL and the examples, Java 6 or Java 7 JDK must be installed on the system, including rudimentary tools such as ant, gcc, make, and python.

The source code of DiSL can be downloaded from the DiSL home page[16], hosted by the OW2 Consortium. In particular, DiSL releases are available at http://forge.ow2.org/projects/disl/files/. After downloading and extracting the latest release of DiSL (i.e., disl-src-2.0.1.tar.bz2 at the time of writing, please consult the README file in the main directory, which provides guidelines for compiling the framework and links to additional documentation.[17]

A very simple example of a DiSL instrumentation can be found in the example/smoke directory. In this example the observed program (i.e., example/smoke/app/src/Main.java) prints a hello-world message, while the instrumentation (i.e., example/smoke/instr/src/DiSLClass.java) inlines the code to print a message at the beginning and at the end of the main method body.

Listing 1 shows the sequence of commands needed to compile the DiSL framework and to run the example. In line 1, we compile the DiSL framework. Line 3 runs the example program.

Listing 1: Compiling the framework and running the included DiSL example.

```
1  [disl]$ ant
2  [disl]$ cd example/app/smoke
3  [disl/example/app/smoke]$ ant run
```

Listing 2 shows the expected output of the instrumented program. Line 2 is the message printed by the observed program, while lines 1 and 3 are the messages printed by the instrumentation.

Listing 2: Output of the included DiSL example.

```
1  Instrumentation: Before method main
```

[16] http://disl.ow2.org

[17] http://disl.projects.ow2.org/xwiki/bin/view/Main/Doc/

```
Application: Inside method main
Instrumentation: After method main
```

It is possible to use the disl.py script to invoke DiSL with user-defined instrumentations, provided the following rules are adhered to:

- All the instrumentation and the analysis classes must be packed into a single jar file, including any external libraries used by the analysis. Such libraries can be added to the jar file using, for example, the jarjar[18] tool.

- The MANIFEST.MF file in the META-INF directory of the jar file must list all the DiSL classes used for the instrumentation; Listing 3 shows the manifest file of the included example. In this case, the instrumentation consists of a single class (i.e., DiSLClass) that can be found in the default package.

Run disl.py -h for information on how to use the script and to list all available parameters.

Listing 3: Manifest file of the included DiSL example.

```
Manifest-Version: 1.0
DiSL-Classes: DiSLClass
```

An archive of all examples and tools presented in this paper (i.e., disl-examples-1.2.tar.bz2) can be downloaded from the DiSL release page. This archive must be extracted within the main directory of DiSL and includes an additional README file that describes how to run the examples. Listing 4 shows how the runExample.sh script can be used to run the example shown in Figure 1 of this paper.

Listing 4: Running the profiler example presented in Figure 1.

```
[disl/examples]$ ./runExample.sh \
2.1-Method_Execution_Time_Profiler-Figure_1
```

[18] http://code.google.com/p/jarjar/