# Introduction to Dynamic Program Analysis with DiSL

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

Yudi Zheng
University of Lugano
Lugano, Switzerland
first.last@usi.ch

Danilo Ansaloni
University of Lugano
Lugano, Switzerland
first.last@usi.ch

Lubomír Bulej
Charles University
Prague, Czech Republic
lubomir.bulej@d3s.mff.cuni.cz

Aibek Sarimbekov
University of Lugano
Lugano, Switzerland
first.last@usi.ch

Walter Binder
University of Lugano
Lugano, Switzerland
first.last@usi.ch

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

## ABSTRACT

DiSL is a new domain-specific language for bytecode instrumentation with complete bytecode coverage. It reconciles expressiveness and efficiency of low-level bytecode manipulation libraries with a convenient, high-level programming model inspired by aspect-oriented programming. This paper summarizes the language features of DiSL and gives a brief overview of several dynamic program analysis tools that were ported to DiSL. DiSL is available as open-source under the Apache 2.0 license.

## Categories and Subject Descriptors

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

## Keywords

Dynamic program analysis, bytecode instrumentation, aspect-oriented programming, domain-specific languages, Java Virtual Machine

## 1. INTRODUCTION

Dynamic program analysis (DPA) techniques enable essential software engineering tools, such as profilers, debuggers, and testing tools. DPA tools often rely on code instrumentation, i.e., the insertion of analysis code at selected locations in the analyzed program. As nowadays many programming languages are compiled to bytecode for the Java Virtual Machine (JVM), bytecode instrumentation has become a common technique for building DPA tools for the JVM.

JVM bytecode instrumentation is supported by a variety of bytecode engineering libraries, such as ASM, BCEL, Javassist, or ShrikeBT, to mention some of them. Generally, these tools offer a rather low level of abstraction, which results in high development effort for building DPA tools; the sources of such tools tend to be verbose and difficult to maintain and to extend. On the other hand, bytecode manipulation libraries give the expert developer many opprtunities to optimize the inserted code, enabling efficient DPA tools.

Some researchers have proposed the use of aspect-oriented programming (AOP) for creating DPA tools. Thanks to its pointcut/advice mechanism, AOP offers a high abstraction level that allows to concisely specify certain kinds of instrumentations. Consequently, the use of AOP promises to significantly reduce the development effort for building DPA tools. Unfortunately, mainstream AOP languages such as AspectJ lack join points at the level of e.g. basic blocks of code or individual bytecodes that would be needed for certain DPA tools. In addition, aspect weavers often introduce high overhead due to complex code transformations involved. For example, in AspectJ, the access to dynamic reflective join point information may involve object allocations and iterations (e.g., to store the arguments of a method in an array) that are not visible in the instrumentation code.

DiSL [1, 3, 2] is a new domain-specific language for JVM bytecode instrumentation that combines the strengths of low-level bytecode manipulation libraries (i.e., expressiveness and efficiency) with the strength of AOP (i.e., high-level programming abstractions). DPA tools written in DiSL tend to be as concise as equivalent tools written in a mainstream AOP language (if this is possible). However, in contrast to prevailing AOP languages, DiSL features an open join point model where any region of bytecodes in a program can become a join point. Moreover, the DiSL weaver produces efficient code that does not incur any unexpected costs; the weaver introduces neither object allocations nor loops that were not written by the programmer.

Below we give a brief summary of DiSL's language features and mention some DPA tools that we successfully ported to DiSL.

## 2. LANGUAGE FEATURES

DiSL is a language hosted in Java and uses annotations to direct the insertion of the code. The language follows the Aspect-oriented paradigm, but is tailored specifically for bytecode instrumentation. To the developer, DiSL provides several key constructs that we now review in turn.

Marker is a DiSL construct responsible for capturing the join points and defining a bytecode region where the selected

*ICPE'13,* April 21–24, 2013, Prague, Czech Republic.
ACM 978-1-4503-1636-1/13/04.

instrumentation will be applied. For common tasks, DiSL provides a predefined set of markers that allow instrumenting method body, basic blocks of code, method invocations, or single bytecodes. However, a developer can take advantage of the open join point model of DiSL and implement markers that intercept arbitrary bytecode sequences.

To provide control over the selection of classes and methods for instrumentation, DiSL supports coarse-grained join-point filtering using a simplified scoping language, and fine-grained filtering using guards. The guards are written in Java, enabling the developer to express complex join-point filtering conditions evaluated at instrumentation time.

Another two constructs provide efficient data passing among the instrumented locations. The first is called Synthetic-local variable and allows passing data in the context of the instrumented method. The second, called Thread-local variable, provides thread local storage.

The mandatory feature of all instrumentation languages is access to static and dynamic context information. In DiSL, static context information is accessed through a library of predefined static context classes. The classes provides various information about the currently instrumented method or class. This includes method name, method visibility, method descriptor, class name, class signature or name of the super class. The developer is free to define custom static context classes, e.g., to provide access to the results of a custom static analysis. The static context data is inserted directly into the bytecode as a constants. This implies that static context information can only come in form of Java basic types or Strings.

DiSL also provides access to the dynamic context information, which includes the *this* object, method arguments and local variables, and operands on the stack.

DiSL is designed for application observation only. It effectively prevents the instrumentation from changing the application control flow. Besides prohibiting the insertion of the return or jump instructions, it also ensures that no uncaught exception escapes the instrumentation.[1]

Finally, DiSL enables full bytecode coverage and avoids disturbing the instrumented application. This is achieved by instrumeting the application classes in a separate JVM.[2]

## 3. RECASTED TOOLS

Many dynamic program analysis tools for the JVM have already been recasted in DiSL. We found that for tools that were originally implemented with a low-level bytecode manipulation library, the recasted tools were much more concise than the original versions. Regarding startup performance, the recasted tools were slightly slower, and concerning steady-state performance, they reached the same level of performance as the original versions. For tools that were originally implemented in a mainstream AOP language, the recasted tools were about the same size, but significantly outperformed the original versions.

Amongst others, we recasted the following tools with DiSL: **Cobertura**[3] is a tool for Java code coverage analysis. At runtime, Cobertura collects coverage information for every line of source code and for every branch. **EMMA**[4] is another coverage analysis tool for Java. At runtime, EMMA collects coverage information for every basic block of every method in every class. **HPROF** is a heap and CPU profiler for Java distributed with the HotSpot JVM. **JCarder**[5] is a tool for finding potential deadlocks in multi-threaded Java applications. **JP2**[6] is a calling-context profiler for Java that keeps the execution trace of the application in a calling-context tree. **JRat**[7] is a call graph profiler for Java. For each method, JRat collects the execution time of each invocation, grouped by the caller. **RacerAJ**[8] is a tool for finding potential data races in Java applications. **ReCrash**[9] is a tool for reproducing software failures. **TamiFlex**[10] is a tool that helps other (static analysis) tools deal with reflection and dynamically generated classes in Java.

## 4. CONCLUSION

DiSL is a domain-specific language for instrumentation-based dynamic program analysis of applications running in the JVM. It offers high-level language constructs tailored for dynamic analysis tasks. The design of DiSL has been inspired by AOP. However, in contrast to mainstream AOP languages, DiSL features an open join point model where any bytecode region can become a join point. DiSL builds on top of the popular bytecode manipulation library ASM. DiSL also offers interfaces for framework extensions based on ASM. DiSL is available open-source under the Apache 2.0 license on OW2: http://disl.ow2.org/

*Acknowledgements:* The research presented in this paper has been supported by the Swiss National Science Foundation (project CRSII2_136225), by the Scientific Exchange Programme NMS–CH (project code 11.109), by a Sino-Swiss Science and Technology Cooperation (SSSTC) Exchange Grant (project no. EG34–092011) and Institutional Partnership (project IP04–092010), by the European Commission (Seventh Framework Programme grant 287746), by the Czech Science Foundation (project GACR P202/10/J042), and International Cooperation Program of Shanghai (No.11530700500).

## 5. REFERENCES

[1] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: A Domain-specific Language for Bytecode Instrumentation. In *Proc. 11th Intl. Conf. on Aspect-oriented Software Development (AOSD'12)*, pages 239–250. ACM, 2012.

[2] L. Marek, Y. Zheng, D. Ansaloni, A. Sarimbekov, W. Binder, P. Tůma, and Z. Qi. Java Bytecode Instrumentation Made Easy: The Disl Framework for Dynamic Program Analysis. In *Proc. 10th Asian Symposium on Programming Languages and Systems (APLAS 2012)*, pages 256–263. Springer, 2012.

[3] Y. Zheng, D. Ansaloni, L. Marek, A. Sewe, W. Binder, A. Villazón, P. Tuma, Z. Qi, and M. Mezini. Turbo DiSL: Partial Evaluation for High-Level Bytecode Instrumentation. In *Proc. 50th Intl. Conf. on Objects, Models, Components, Patterns (TOOLS'12)*, volume 7304 of *LNCS*, pages 353–368. Springer-Verlag, 2012.

[1]This feature can be deactivated if required.

[2]A native agent is used in the application JVM to exchange data with the instrumentation JVM.

[3]http://cobertura.sourceforge.net/

[4]http://emma.sourceforge.net

[5]http://www.jcarder.org/

[6]https://code.google.com/p/jp2/

[7]http://jrat.sourceforge.net/

[8]http://www.bodden.de/tools/raceraj/

[9]http://groups.csail.mit.edu/pag/reCrash/

[10]https://code.google.com/p/tamiflex/