Science of Computer Programming 98 (2015) 100–115

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

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*

ARTICLE INFO

Article history:
Received 2 November 2012
Received in revised form 21 December 2013
Accepted 7 January 2014
Available online 15 January 2014

Keywords:
Dynamic program analysis
Bytecode instrumentation
Aspect-oriented programming
Domain-specific languages
Java Virtual Machine

ABSTRACT

Dynamic program analysis (DPA) tools assist in many software engineering and development tasks, such as 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.

© 2014 Elsevier B.V. All rights reserved.

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, such as 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, such as profiling [1,2], debugging [3–6], 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

* Corresponding author.
E-mail addresses: lukas.marek@d3s.mff.cuni.cz (L. Marek), yudi.zheng@usi.ch (Y. Zheng), danilo.ansaloni@usi.ch (D. Ansaloni), lubomir.bulej@usi.ch (L. Bulej), aibek.sarimbekov@usi.ch (A. Sarimbekov), walter.binder@usi.ch (W. Binder), petr.tuma@d3s.mff.cuni.cz (P. Tůma).

0167-6423/$ – see front matter © 2014 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2014.01.003

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 [22–24], 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 [22–24]. 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.

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

[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.