Live probes for free

Tobias Pape[a], Cristina V. Lopes[b], and Robert Hirschfeld[a]

a Hasso Plattner Institute, University of Potsdam, Germany

b University of California, Irvine, USA

Abstract *Context* In his presentation "Inventing on Principles", Bret Victor demonstrates a live code editor: by specifying input values for a function, we can observe in real time the values taken by the variables during execution, as the code is written. This information is often obtained using a language designed for live programming or by instrumentation of a specific runtime. *Inquiry* . . . *Approach* In this paper we propose to exploit the capabilities of debuggers to obtain the data needed to design a live code editor. *Knowledge* . . . *Grounding* . . . *Importance* . . .

ACM CCS 2012

- **General and reference** → *Computing standards, RFCs and guidelines*;
- **Applied computing** → **Publishing**;

Keywords programming journal, paper formatting, submission preparation

The Art, Science, and Engineering of Programming

Perspective The Art of Programming

Area of Submission Social Coding, General-purpose programming

1 Introduction

2 Problem Overview

3 Using the Debugger ?

3.1 Stack Recording

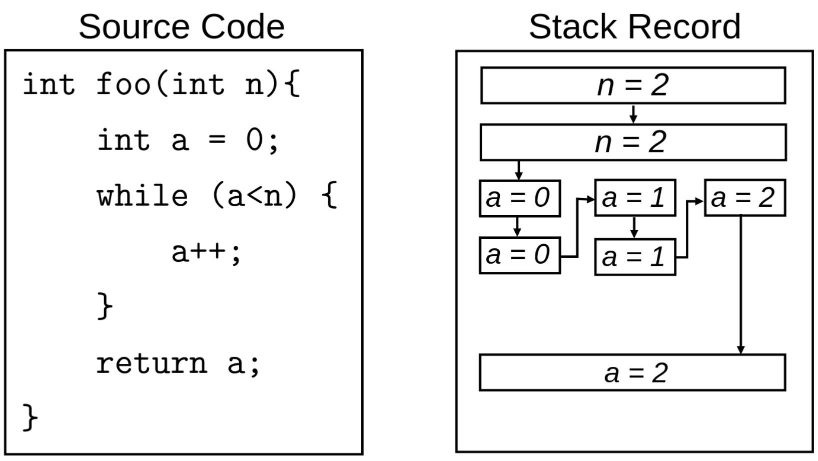

Figure 1 Stack Recording example

In order to generate the data needed to probe a variable, we need to be able to retrieve the state of the variable during execution. This information is contained in the stackframe at the time of execution. However, we need to add spatial and temporal information to this: we need to associate each stackframe with the location in the code at which it was retrieved, and we also need to know the order in which these stackframes were retrieved.

In this paper, we introduce a structure for representing this data: a *stack recording* 1. A stack trace represents the different stackframes in the form of a chain, where each stackframe has a reference to the corresponding line of source code. This representation allows us to maintain a link between the spatial location (the reference to the source code) and the temporal location (the order of these stackframes) of the execution. This representation has several advantages: it is easy to construct from debugger information, and it applies to most programming languages.

3.2 Keep Alive Agent

Figure 2 Keep Alive Agent

A live environment must be able to react to two different events: a change in the code or a change in the test/input data. If the code changes, we need to re-execute the code, and in the case of a compiled language, we need to compile the new code first. If the data changes, we need to be able to re-execute the programme with the new data. In addition, it is necessary to keep compilation and execution times low enough to maintain an interactive experience with the user.

To meet these constraints, we propose to use the debugger on an intermediate program, a *keep-alive agent*. This program keeps the debugger alive between executions and code changes to reduce initialisation times. The general operation of a keep-alive agent is shown in Figure 2 :

- At the start of the session, the debugger is started on the agent. Once initialised, the agent is paused.
- The target code is loaded into the agent and a breakpoint is set at the input of the target method.
- Execution of the target method triggers the breakpoint at the input of the method, and execution continues step by step to build the *stack recording*.

4 Live Probes in Java with JDI

We initially developed a Java backend using JDI, a debugging interface for Java, to implement the concepts discussed in the previous section. The keep-alive agent includes a method for loading classes into the JVM.

When executing the target method, the arguments are created in the client JVM and then passed to the debugger JVM using the `mirrorOf` and `newInstance` methods from the reflection API. The target method is invoked using `invokeMethod`. To handle events from the JDI when breakpoints are set, a dedicated thread is used to prevent deadlocks.

If modifications are needed in the code of the target method, we employ the `redefineClasses` method. This method allows changing the content of a class loaded in the JVM using an array of byte codes. However, it has a limitation: it only works if the class signature remains unchanged. If the class structure is modified, restarting the debugger is necessary.

5 Generalizing Live Probes with Debugger Adapter Protocol

6 Evaluation

7 Related Work

8 Conclusion

About the authors

Tobias Pape is the author of this LaTeX class. Contact him at tobias.pape@hpi.uni-potsdam.de.

Cristina V. Lopes is associate editor for the first two issues of The Art, Science, and Engineering of Programming. Contact her at lopes@ics.uci.edu.

Robert Hirschfeld is chair of the AOSA steering committee. The Art, Science, and Engineering of Programming is published by AOSA. Contact Robert at hirschfeld@hpi.uni-potsdam.de.