Stepwise Back-in-time Debugging

Vasily Kirilichev[a], Eric Seckler[a], Benjamin Siegmund[a],
Michael Perscheid[b], and Robert Hirschfeld[b]
Hasso Plattner Institute, University of Potsdam, Germany
[a]{firstname.lastname}@student.hpi.uni-potsdam.de,
[b]{firstname.lastname}@hpi.uni-potsdam.de

Abstract: To fully understand how observable failures come into being, back-in-time debuggers provide access to past executions. However, the required run-time analysis is often associated with an inconvenient overhead that renders current tools impractical for frequent use. Potentially large execution histories are expensive to collect and include much data that needs to be analyzed.

Our previously presented stepwise run-time analysis speeds up this analysis by dividing it into multiple steps according to user interaction: A high-level analysis of the method call tree followed by on-demand refinements of object states. This paper advances our approach with a statement-level refinement that allows developers to *step* through large execution histories in forward and backward direction. The corresponding extension of PathFinder, our lightweight back-in-time debugger, provides for instant access to relevant run-time data without collecting needless data up front.

1 Introduction

To better comprehend what causes failures, back-in-time debuggers [Lew03] can help developers by providing access to all required execution details. These debugging tools include every information for describing what happened before observable failures. However, traditional dynamic analysis techniques such as post-mortem debuggers [Lew03] are typically inefficient, time-consuming, and impractical for frequent use. Most approaches capture comprehensive information about the entire execution up-front so that required run-time analysis is often associated with an inconvenient overhead.

Our stepwise run-time analysis [PSH$^+$10] as basis for our lightweight back-in-time debugger named PathFinder [Per13] enables an experience of immediacy that current tools are missing by capturing run-time data only when needed. Low cost can be achieved by dividing the program analysis into multiple runs according to user interaction[1]. Developers immediately retrieve a shallow overview of the execution history that is expanded with user-relevant object states step by step.

Even if our approach enables immediacy characteristics during exploring execution histories, so far it is limited to the method-level. For that reason, this paper presents an extension to our stepwise run-time analysis that enables *stepping* at the statement-level on

[1] We leverage test cases as deterministic entry points into run-time behavior in order to reproduce arbitrary points in a program execution.

demand. Similar to symbolic debuggers, developers can step into and over a statement as well as to its return point, but also back to the previous statement and to the sender. With our PathFinder extension, developers now experience a complete back-in-time debugger which still includes the immediacy characteristics of our original approach.

The contributions of this paper are as follows[2]:

- An extension of our previous stepwise run-time analysis [PSH+10] that allows developers to immediately *step* through statements in execution histories in both forward and backward direction.
- An implementation of this approach as part of our lightweight back-in-time debugger called PathFinder [Per13].

2 Stepping Back and Forth in an Execution History

We describe how we can apply the stepping functionality known from symbolic debuggers to execution histories, providing developers with a familiar interface to navigate the runtime behavior. We present five different stepping actions to developers: *step into*, *step over*, *step return*, *step back*, and *step to sender*. Our approach uses a call tree data structure as representation of the execution history. It represents calling relationships between executed methods. In it, each method call appears as one node. For the stepping functionality, we keep track of the active statement within one of these method call nodes, which can then be highlighted accordingly in the visual representation of the call tree.

The *step into*, *step over* and *step return* functionality is very similar to conventional debuggers. If the currently active statement in the call tree is a call to a method, *step into* sets the first statement in the node of this method call as active statement. *Step over* evaluates statements as a unit and steps to the statement succeeding the active statement in its node. If no further statements exist in the active node, the effect of *step over* equals that of *step return*. *Step return* jumps back to the parent node of the active node and into the statement in the parent node that follows the method call statement corresponding to the active node.

For execution histories, it is possible to add two additional stepping operations. *Step back* is the exact opposite to *step over*. It steps to the statement preceding the active statement. In the case that no further preceding statements exist in the active node, its effect equals that of *step to sender*. *Step to sender*, in turn, is the opposite to *step into*, and is similar to *step return*, but jumps back to the exact method call statement in the parent node of the active node that corresponds to the active node.

---

[2] A screencast can be found at: `http://www.youtube.com/watch?v=5FtSaZtbUXg#t=210`

3 Stepwise Run-time Analysis at Statements

Our extension of the stepwise run-time analysis follows the same strategy as our original work [PSH$^+$10]. We collect the information necessary for the execution of the stepping actions described before incrementally and on demand. In order to execute the stepping actions, we need to know the sequence of statements executed in the respective call nodes as well as the child nodes that these are corresponding to. Then, we can identify the statement preceding or following another one (necessary for *step over* and *step back*), the child node a method call statement is corresponding to (for *step into*), and the statement in a parent node corresponding to a specific child node (for *step return* and *step sender*).

PathFinder analyzes the execution history by instrumenting the code using method wrappers. We extend PathFinder with a new method wrapper, which is activated only for the refinement analysis necessary to gather the information mentioned above, and only for the method call under investigation. Instead of simply executing the wrapped method, the wrapper starts a simulated interpretation of the Smalltalk method source. This way, it is able to record the order of the individual statements actually executed. Additionally, we wrap called methods and record which statement is currently active in their calling method. So, we establish the relationship between the executed statements in the parent node and its child nodes. The execution of other methods in the call tree is not affected by this selective analysis. Whenever we step into a new method that we did not perform this analysis before, we immediately perform our analysis again for the new node in background.

4 Using the Stepping Mechanisms in PathFinder

Figure 1 shows our PathFinder back-in-time debugger displaying the call tree of a failing test case in the DicThesaurusRex Hunspell library[3]. The assertion in *testSpell* fails because of a bug in the method *DTRCamelCaseParser>>readString*. We use this screenshot to explain the stepping functionality we added to PathFinder.

The stepping analysis can be initialized from any method in the call tree via the *step into call node* button (f). This button initializes the active stepping statement to the first statement in the selected node. The user can then use the buttons in the toolbar to *step over* (a), *step into* (b), *step return* (c), *step back* (d) and *step to sender* (e). The target statements of these actions are illustrated with arrows labeled with the respective upper case letters.

We highlight the active stepping node visually by changing its background color to a darker blue; the active statement is highlighted through text selection (lighter blue background). Additionally, we also highlight the child node corresponding to the active statement (if one exists) with a lighter blue background. In the figure, this is the *readString* call node.

During our initial informal user studies, we could validate that the incremental stepping analysis does not cause any interaction delays noticeable by user. We have been using the feature in a number of existing Smalltalk projects without any limitations.

[3] https://www.hpi.uni-potsdam.de/hirschfeld/trac/SqueakCommunityProjects/wiki/dicThesaurusRex

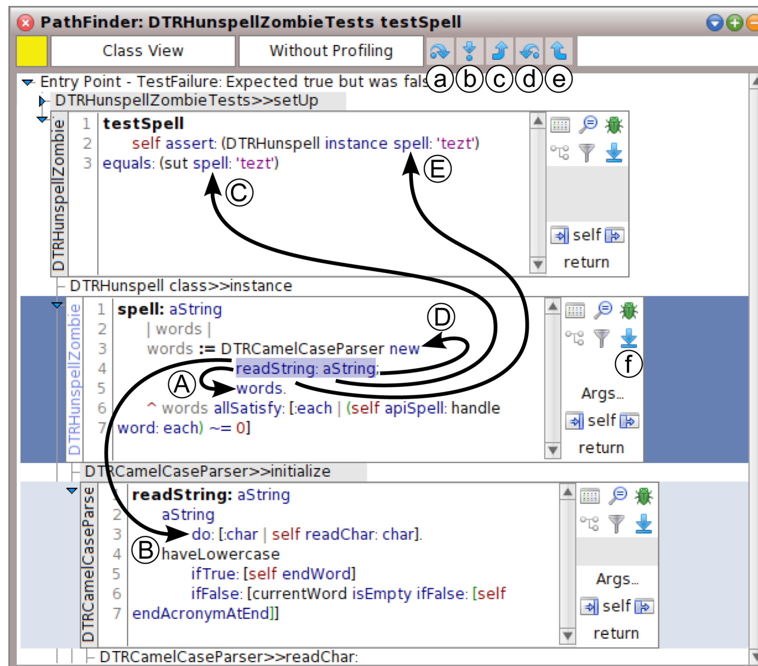

Figure 1: Stepping actions in PathFinder.

5 Conclusion

We have extended our stepwise run-time analysis to support stepping through execution histories. Using an incremental dynamic analysis, we are able to process large execution data step by step, while preserving the immediacy characteristics for developers. This approach makes our back-in-time debugger practical for frequent use. One possible area of future work is to allow developers to change methods directly in the call tree. Therefore, we will investigate adding support for *hot recompilation* to PathFinder.

References

[Lew03] B. Lewis. Debugging Backwards in Time. In *AADEBUG*, pages 225–235, 2003.

[Per13] M. Perscheid. *Test-driven Fault Navigation for Debugging Reproducible Failures*. PhD thesis, Hasso-Plattner-Institute, University of Potsdam, 2013.

[PSH+10] M. Perscheid, B. Steinert, R. Hirschfeld, F. Geller, and M. Haupt. Immediacy through Interactivity: Online Analysis of Run-time Behavior. In *WCRE*, pages 77–86, 2010.