Follow the Path: Debugging Tools for Test-Driven Fault Navigation

Michael Perscheid and Robert Hirschfeld
Software Architecture Group
Hasso-Plattner-Institute
University of Potsdam, Germany
firstname.lastname@hpi.uni-potsdam.de

***Abstract*—Debugging failing test cases, particularly the search for failure causes, is often a laborious and time-consuming activity. Standard debugging tools such as symbolic debuggers and test runners hardly facilitate developers during this task because they neither provide advice to failure causes nor back-in-time capabilities.**

In this paper, we present test-driven fault navigation as a debugging guide that integrates spectrum-based and state anomalies into execution histories in order to systematically trace failure causes back to defects. We describe and demonstrate our Path tools that implement our debugging method for the Squeak/Smalltalk development environment.

***Index Terms*—Debugging, Testing, Back-in-time, Spectrum-based Anomalies, Likely Invariants, Squeak, Smalltalk**

I. Introduction

The correction of software failures tends to be very cost-intensive because their debugging is an often time-consuming development activity. During this activity, developers largely attempt to understand what causes failures: Starting with a test case that reproduces the observable failure, they have to follow failure causes along the infection chain back to the root cause (defect). This idealized procedure requires deep knowledge of the system because failures and defects can be far apart. Unfortunately, common debugging tools are inadequate for systematically investigating such infection chains in detail [1].

Our *test-driven fault navigation* [2] is a debugging guide that integrates spectrum- [3] and state-based [4] anomaly detection into a systematic breadth-first search for tracing failure causes back to defects. Starting with at least one test case that can reproduce the observable failure, we localize anomalies by comparing the method coverage and occurred state properties of all failed and passed test cases. For example, methods that are being executed by a large number of failing but only a few passing tests have a higher failure cause probability (anomaly) than methods that are being executed by less failing but many passing test cases. By integrating such anomalies into a back-in-time debugger [5], we highlight suspicious method calls, reveal corrupted state properties, and allow developers to distinguish between suspicious and expected run-time behavior. Thus, our execution histories include additional information about failure cause probabilities that gives developers helpful advice on how to follow infection chains back to their root causes.

Our systematic search consists of four specific navigation techniques that together support the creation, evaluation, and refinement of failure cause hypotheses. First, *structure navigation* localizes suspicious system parts and restricts the initial search space. Second, *team navigation* recommends experienced developers for helping with failure causes by focusing on authors of anomalies. Third, *behavior navigation* allows developers to follow emphasized infection chains back to root causes. Fourth, *state navigation* identifies corrupted state and reveals parts of the infection chain automatically.

In this paper, we present our *Path tools* that implement test-driven fault navigation for the Squeak/Smalltalk [6]. We demonstrate their application by debugging one example failure step by step until the root cause has been found.

The remainder of this paper is structured as follows: Section II introduces our Path tools that implement our debugging approach. Section III demonstrates a debugging session. Section IV presents related work. Section V concludes.

II. The Path Tools

Our Path tools as presented in Fig. 1 primarily consist of two debugging tools that realize test-driven fault navigation.

A. PathMap: Extended Test Runner Feedback

PathMap [7] is an extended unit test runner for implementing our structure, team, and state navigation. It does not only verify test cases but also localizes failure causes. Its integral components are from left to right a testing control panel (A), a compact tree map visualization of the software system (B), and several flaps for accessing various analysis techniques (C).

The testing control panel (A) provides nearly the same functionality as a standard test runner. Developers can choose from different test suites of the selected project and run them.

The tree map (B) in the middle presents the structure of the system under observation with its packages, classes, and methods. Each small method box can be colored to represent test case and analysis results. It is possible for developers to interact with the tree map: receive the name of a source code entity by hovering over a box; request additional information about colored metric values (as shown in the message box); or debug into a suspicious method execution with our lightweight back-in-time debugger called PathFinder. Furthermore, the test runner also presents a status bar on the top displaying a

978-1-4799-3752-3/14/$31.00 © 2014 IEEE

CSMR-WCRE 2014, Antwerp, Belgium
Tool Demonstration

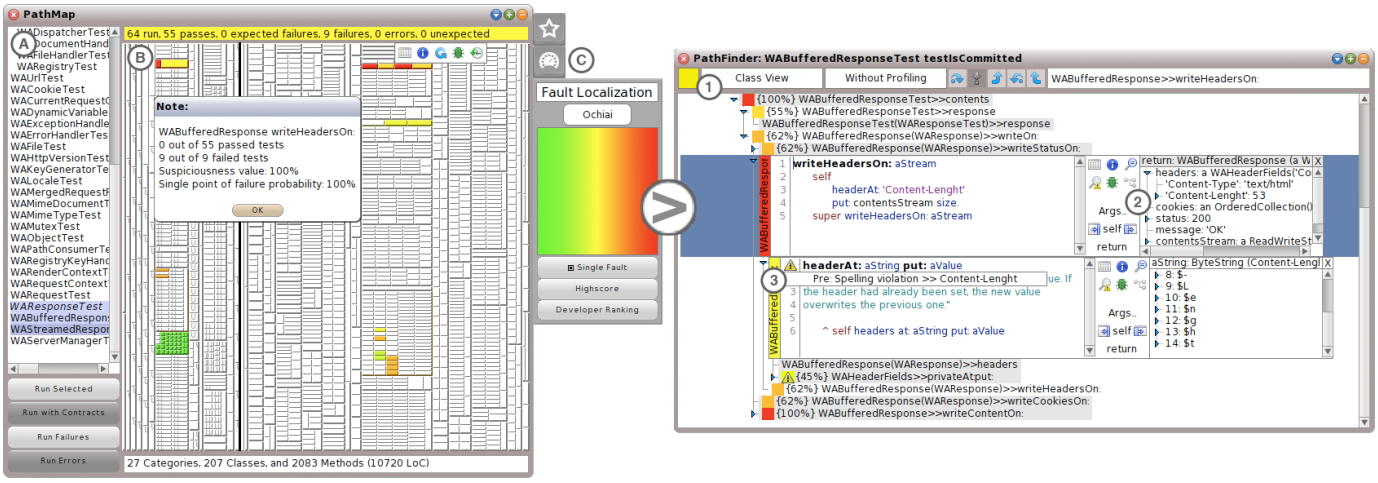

Fig. 1. PathMap as an extended test runner and PathFinder as a lightweight back-in-time debugger implement our test-driven fault navigation.

summary of the test suites execution and a status bar on the bottom displaying a summary of project metrics.

Flaps on the right (C) set PathMap into specific analysis modes for collecting valuable feedback during the execution of test cases. Without an opened flap, PathMap acts like a standard test runner and only colors test case results in the tree map. In Fig. 1 (left), the fault localization flap is open and allows developers to start the structure navigation [8]. If developers run selected tests, we automatically record their coverage, compute spectrum-based anomalies [3], and color the map with suspiciousness and confidence scores. Within the flap, developers can choose a proper spectrum-based metric, see a color legend, and enable filtering of partially covered methods. For our team navigation [8], pushing the "Developer Ranking" button presents a ranked list of expert developers by relating suspicious methods with their authors. Moreover, we offer other flaps to further reveal hidden test knowledge. For example, the inductive analysis flap allows developers to derive likely invariants [4] for our state navigation [9].

B. PathFinder: Lightweight Back-in-Time Debugger for Tests

PathFinder [10] is our lightweight back-in-time debugger [5] for exploring specific test case executions with a special focus on fault localization. Not only does it provide immediate access to run-time information [10], but also classifies traces with suspicious behavior [8] and anomalous state [9]. Developers start PathFinder at a chosen method in PathMap and follow the infection chain back to its root cause. Its main components are a control panel on the top and the test case execution history.

In Fig. 1 (right), PathFinder provides a control panel (1) to set up the dynamic analysis of test cases and to support the navigation through the large amount of run-time data. The yellow box presents the final test result and allows developers to rerun test cases. Views and profiling information enhance the results of initial call trees. While views record more details about called receiver objects such as names and identities, profiling precisely measures the required time for executing a test case. Finally, the subsequent buttons and a query engine provide functionality for arbitrarily navigating through an execution history.

At the bottom of Fig. 1 (right), the visualized information primarily consists of a call tree that reflects a particular test case run. A call tree provides comprehensive information of the entire program execution and shows how methods call each other. From top to bottom, each node represents one method call and their subtrees describe its called methods. Each method call node consists of a colored box with a percentage value for its suspiciousness score and a name representing receiver class, implementation class in parentheses, and method name. We provide arbitrary navigation through method call trees and their state spaces (i. e. object properties). Developers can follow traces in both directions and expand and collapse subtrees interactively.

Some of the tree nodes have been expanded in Fig. 1 (right) to reveal details about the method implementation and the applied state. For example, an expanded method call node (2) shows its source code, a control panel for requesting details and refining run-time data, and the return value. Most notably, the control panel allows developers to start a source code editor and a symbolic debugger, obtain additional information about anomalies, refine coverage and spectrum-based fault navigation at statements, and explore object states. After indicating interest in a specific argument, receiver, or return object, we reexecute the test case, make a deep copy of the requested object, and present it in an object explorer on the right. Developers can explore all object properties and compare them to other method nodes in the execution history. Furthermore, our state navigation [9] maps violated contracts to traces by adding small exclamation marks (3) to methods. Developers can further inspect such state anomalies and receive detailed information about the exact violation.

III. Debugging Demonstration

To demonstrate our test-driven fault navigation and its corresponding Path tools, we debug failure 4 of our previous

user study [8] step-by-step[1]. As the underlying system, we choose Squeak's iCalendar library[2] which implements the identically named file format for sharing meeting requests and tasks independent of specific calendar applications. Unfortunately, iCalendar's synchronization mechanism does not work as expected and obsolete calendar events remain active even if users deleted them. We have already reproduced this observable failure with one failing test case.

A. Step 1: Structure Navigation

Before we can start with our test-driven fault navigation, we have to declare the system under observation. For that reason, we create a new Path project for the iCalendar library and choose its corresponding source code packages. After that, our corresponding analyses are limited to this specific project and we are ready for debugging with our approach.

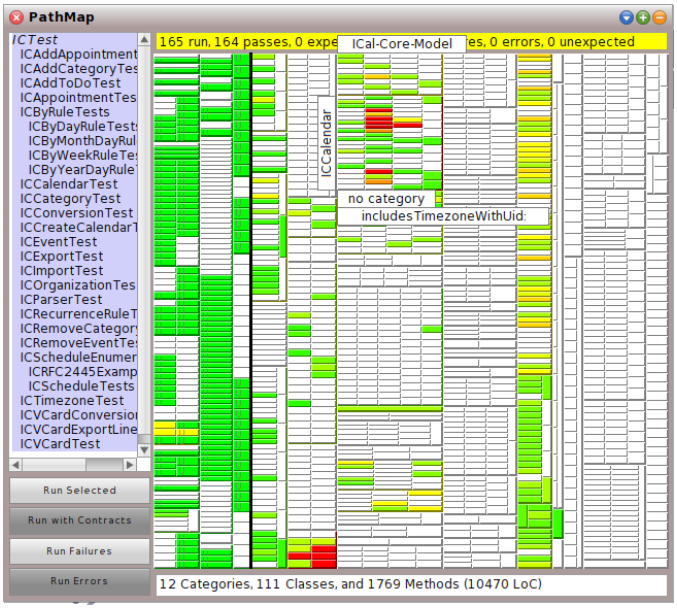

Fig. 2. Step 1: Where to start debugging? Our structure navigation localizes two very suspicious classes (red colored areas in `ICCalendar` and `ICImporter`) for creating initial failure cause hypotheses.

In the first step, our structure navigation helps us to emphasize suspicious system parts and to create a first failure cause hypothesis. We start our extended test runner PathMap, open the "fault localization" flap, and run all test cases of iCalendar. In doing so, our tool compares the coverage between all passing and failing test cases and so reveals suspicious methods or in other words anomalies. Fig. 2 presents the results within our compact tree map. The more red a method is, the higher is the failure cause probability (more failing than passing tests are involved during the execution of this method). This heat map allows us to get a first advice about what is going wrong within the iCalendar system. In this example, we see two hot spots at the class `ICImporter` and `ICCalendar`. However, it is still unclear, which of these methods includes the defect, how these anomalies are related to each other, and how the failure comes into being.

B. Step 2: Team Navigation

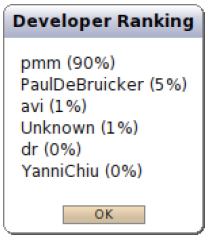

Fig. 3. Step 2: Who understands failure causes best? Based on suspicious methods, our team navigation identifies several developers and a very experienced one for debugging the failure.

As we are not completely familiar with the iCalendar project, it can be very valuable to ask a more experienced developer for help. Our optional team navigation proposes a novel metric that computes expert knowledge by combining anomalies with their corresponding authors. As these suspicious methods have a high probability to include failure causes, their authors are more likely to understand and debug the failure. Compared to expertise metrics that consider the entire system, we restrict the search space on system parts that are related to failure causes only. After clicking the button "Developer Ranking" in PathMap's "fault localization" flap and choosing a proper author metric, we get a prioritized list of author initials. Fig. 3 proposes "pmm" as a developer with the best knowledge about the two suspicious classes. Please note: This metric does not blame developers, it rather recommends experts for helping with failures.

C. Step 3: Behavior Navigation

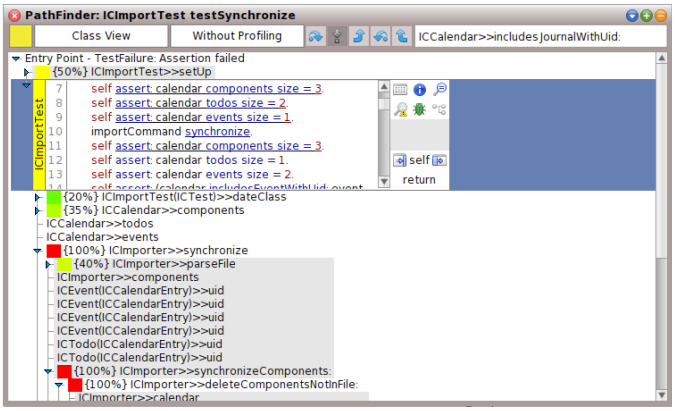

Fig. 4. Step 3: What happened before failures? Our behavior navigation classifies the complete execution history of a failing test case with suspicious methods (colored method calls) and so guides developers back to root causes.

Based on the results of our structure navigation, we would like to understand how the failure comes into being—we need access to the entire execution history of the failing

[1]More information on our test-driven fault navigation can be found at: http://www.michaelperscheid.de/projects/

[2]http://www.squeaksource.com/ical/

test case. For that reason, we start our lightweight back-in-time debugger PathFinder by clicking on the failing test case in our PathMap visualization. After that, a new debugger window immediately opens and presents the method call tree of the failing `testSynchronize` as shown in Fig. 4. The colors represent suspicious method calls (reused anomalies from our structure navigation) that help us to navigate through the large amount of run-time information. The more red a method call is, the more likely it includes a failure cause. Hence, we can follow all red methods and understand how these anomalies are related to each other. Starting with our failing test case at the top, the next suspicious method seems to be `ICImporter>>synchronize`, looking at its sub-calls `synchronizeComponents:` is also very suspicious and so on. Following these anomalies allows us to abbreviate a lot of methods that are not related to our failure and so allows us to reduce the required debugging effort. Unfortunately, we still have to understand potential failure causes and corrupted state properties by ourselves.

D. Step 4: State Navigation

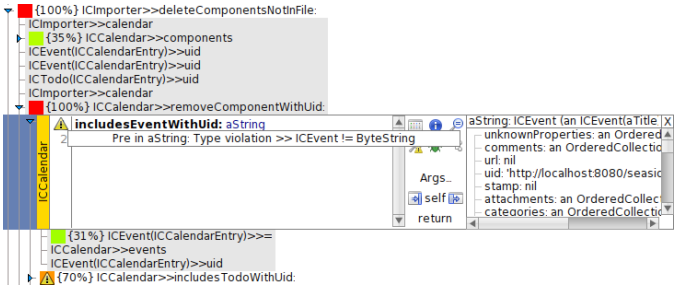

Fig. 5. Step 4: Which state properties are infected? Our state navigation automatically identifies unexpected objects and so reveals parts of the infection chain. Here, we identify a wrong (`ICEvent`) type instead of a `String`.

In the last step of our test-driven fault navigation, we automatically identify unexpected state properties, map them to the execution history of our failing test, and find out what is going wrong. First, we have to learn expected state properties from the remaining passing test cases. We start PathMap again, open the "induction" flap, choose a harvester mode, and run all tests. In doing so, PathMap runs all passing test cases, observes their applied objects, and derives generalized object properties. Based on these results, we add dynamic contracts that trigger violations if one of the identified invariants does not hold. Now, PathFinder is able to compare the expected state properties of passing test cases with our failing one. Fig. 5 presents the enhanced method call tree that shows small exclamation marks to highlight such state violations. By hovering over these icons, we get closer information about unexpected state properties. For example, the specific type violation in `ICCalendar>>includesEventWithUid` signalizes a wrong `ICEvent` instead of a `String` object. With that in mind, we can further inspect calling methods (with a focus on suspicious colors and further state anomalies) until we end in the method `deleteComponentsNotInFile`. Here, we find the root cause of our demonstrating failure. This method fails to convert `ICEvent` objects into their proper uid `String` representations. For that reason, the subsequent synchronization mechanism does not find existing objects that should be deleted later on.

IV. Related Work

Test-driven fault navigation and our Path tools are based on the combination and improvement of several debugging techniques. Spectrum-based fault localization [3] provides the basis for our structure navigation. We enhance this idea by reusing anomalies in different perspectives and so give helpful advice on how to follow infection chains back to their root causes. One other perspective is our team navigation that applies such anomalies in order to restrict the search space for suitable experts. A similar idea has been developed in parallel [11]. Our behavior navigation combines such suspicious methods with back-in-time debuggers [5] and, finally, likely invariants [4] emphasize state violations along infection chains. Moreover, to ensure an experience of immediacy when debugging with our tools, we have also developed the so called incremental dynamic analysis [10]. A full discussion of related work can be found in [2].

V. Conclusion

In this tool demonstration, we presented our Path tools that implement our test-driven fault navigation debugging approach. With the help of both PathMap as an extended test runner as well as PathFinder as a lightweight back-in-time debugger, developers are able to answer important questions about failure causes: where to start debugging, who understands failure causes best, what happened before failures, and which state properties are infected?

References

[1] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.

[2] M. Perscheid, "Test-driven Fault Navigation for Debugging Reproducible Failures," Ph.D. dissertation, Hasso-Plattner-Institute, University of Potsdam, 2013.

[3] J. Jones, M. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," in *ICSE*, 2002, pp. 467–477.

[4] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, 2007.

[5] B. Lewis, "Debugging Backwards in Time," in *AADEBUG*, 2003, pp. 225–235.

[6] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself," in *OOPSLA*, 1997, pp. 318–326.

[7] M. Perscheid, D. Cassou, and R. Hirschfeld, "Test Quality Feedback - Improving Effectivity and Efficiency of Unit Testing," in *C5*, 2012, pp. 60–67.

[8] M. Perscheid, M. Haupt, R. Hirschfeld, and H. Masuhara, "Test-driven Fault Navigation for Debugging Reproducible Failures," *Journal of the JSSST on Computer Software*, vol. 29, no. 3, pp. 188–211, 2012.

[9] M. Perscheid, T. Felgentreff, and R. Hirschfeld, "Follow the Path: Debugging State Anomalies along Execution Histories," in *CSMR-WCRE*, 2014.

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

[11] F. Servant and J. Jones, "WhoseFault: Automatic Developer-to-Fault Assignment through Fault Localization," in *ICSE*, 2012, pp. 36–46.