Test-driven Fault Navigation for Debugging Reproducible Failures

Michael Perscheid Michael Haupt Robert Hirschfeld
Hidehiko Masuhara

Debugging failing test cases, particularly the search for failure causes, is often a laborious and time-consuming activity. With the help of spectrum-based fault localization developers are able to reduce the potentially large search space by detecting anomalies in tested program entities. However, such anomalies do not necessarily indicate defects and so developers still have to analyze numerous candidates one by one until they find the failure cause. This procedure is inefficient since it does not take into account how suspicious entities relate to each other, whether another developer is better qualified for debugging this failure, or how erroneous behavior comes to be.

We present *test-driven fault navigation* as an interconnected debugging guide that integrates spectrum-based anomalies and failure causes. By analyzing failure-reproducing test cases, we reveal suspicious system parts, developers most qualified for addressing localized faults, and erroneous behavior in the execution history. The *Paths* tool suite realizes our approach: PathMap supports a breadth first search for narrowing down failure causes and recommends developers for help; PathFinder is a lightweight back-in-time debugger that classifies failing test behavior for easily following infection chains back to defects. The evaluation of our approach illustrates the improvements for debugging test cases, the high accuracy of recommended developers, and the fast response times of our corresponding tool suite.

1 Introduction

Debugging failing test cases tends to be a tedious activity since it requires deep knowledge of the system and its behavior [33]. For localizing failure causes, developers have to follow the *infection chain* backwards; starting from the observable *failure* they follow *erroneous behavior*, including infected state, to the failure-inducing *defect* (root cause) [34]. In practice, however, this process is only partially supported by tools such as symbolic debuggers and test runners. They do not provide advice to *anomalies* in state and behavior (such as deviations from passing test cases), qualified developers for help, or back-in-time capabilities [34]. For this reason, debugging often requires too much time because developers have to rely primarily on their intuition.

To decrease the required debugging effort, *spectrum-based fault localization* [11] reveals anomalies that may help in identifying failure causes. By comparing covered program statements of passed and failed test cases, the approach produces a prioritized list of suspicious statements which restricts the search space and reduces speculations. Unfortunately, anomalies are not failure causes—developers have only a few starting points that must be debugged one by one. By debugging these suspicious source code entities, anomalies and failure causes are not integrated with each other and thus lead to a number of questions that are difficult to answer: what are the relations between anomalies and failures; are there experienced developers

再現可能な誤りをデバッグするためのテスト駆動型誤り発見ツール.
Michael Perscheid and Robert Hirschfeld, Software Architecture Group, Hasso Plattner Institute, University of Potsdam, Germany.
Michael Haupt, Oracle Labs..
増原英彦, 東京大学大学院総合文化研究科, Graduate School of Arts and Sciences, the University of Tokyo.
コンピュータソフトウェア, Vol.29, No.3 (2012), pp.188–211.
[研究論文] 2011 年 10 月 31 日受付.

that understand the failure and its anomalies more easily; how are infected state and anomalous behavior propagated so that failures come to be.

To further reduce debugging costs, we argue that it is important to integrate anomalies and failure causes. An integration supports developers in answering more difficult questions and allows other debugging activities to benefit even from anomalies. Linked views between suspicious source code entities and erroneous behavior help not only to localize causes more efficiently but also to identify the most qualified developers for understanding the current failure.

In this paper, *test-driven fault navigation* integrates anomalies and failure causes to support developers in debugging reproducible failures. Developers isolate suspicious system parts, identify most qualified colleagues for help, and debug erroneous behavior back in time. Based on test cases as descriptions of reproducible failures, we improve spectrum-based fault localization by combining revealed anomalies with a compact system overview, development information, and the execution history. The *Path tool suite* realizes our approach and consists of *PathMap* as an extended test runner that highlights relationships between suspicious system parts and identifies experienced developers for helping with failures and *PathFinder* as a lightweight back-in-time debugger for failing tests that eases navigation to failure causes by emphasizing anomalous behavior. To achieve fast response time for requested run-time information, we base our implementation on step-wise run-time analysis [23]. Our analysis automatically splits and distributes, depending on developers needs, the dynamic analysis over multiple runs and so ensures a high degree of scalability. Thus, we answer quickly where the failure cause is located, which developer is most qualified for fixing the bug, and how erroneous behavior is related to anomalies.

Test-driven fault navigation gives interconnected and fast advice to failure causes by combining test cases, spectrum-based fault localization, and step-wise run-time analysis. The contributions of this paper are as follows:

- *Structural navigation* supports a scalable breadth first search for suspicious system parts based on the results of unit test frameworks[†1] and spectrum-based fault localization. It highlights relationships between anomalies and provides an overview of starting points that are likely to include the failure cause. *PathMap* realizes the structural navigation as an extended test runner. It provides a scalable tree map visualization and a low overhead analysis framework that computes anomalies at methods and refines results at statements on-demand.
- *Team navigation* identifies other developers as experts for helping with failures based on spectrum-based anomalies. Our metric restricts the set of possible experts to suspicious methods only with the result that our suggested developers understand anomalies best. Assuming that anomalies have a high probability to include failure causes, we are able to recommend experts even though the defect is still unknown.
- *Behavioral navigation* follows the infection chain backwards from the observable failure to the past defect. With the help of step-wise run-time analysis and spectrum-based fault localization, we realize a lightweight back-in-time debugger for test cases that highlights anomalies for easier navigation through erroneous behavior. *PathFinder* allows immediate access to run-time information and supports developers in understanding the relationship between anomalies and causes.

We have implemented test-driven fault navigation as part of our *Paths* framework that provides integrated tool support for the Squeak/Smalltalk IDE. Moreover, we illustrate our approach with a case study and evaluate it with respect to improvements for debugging, accuracy of our developer ranking metric, and efficiency of our tool suite.

The remainder of this paper is structured as follows: Section 2 introduces our motivating case study and explains contemporary challenges in testing and debugging. Section 3 presents the integration of anomalies and failure causes and how our test-driven fault navigation influences the debug-

†1 We consider unit test frameworks, for instance xUnit [3], as a technique for implementing different kinds of test cases. Our approach works for combined testing including among others acceptance, integration, and module tests.

ging process. Sections 4, 5, and 6 explain the structural, team, and behavioral navigation in detail. Section 7 outlines the implementation of our Paths framework and discusses the introduction of our approach. Section 8 evaluates the improvements, accuracy, and efficiency of our approach. Section 9 discusses related work and Section 10 concludes.

2 Finding Causes of Reproducible Failures

We introduce a motivating example for an error taken from the Seaside Web framework [24] that serves as a basis for our discussion of challenges in testing and debugging. Seaside is an open source Web framework written in Smalltalk and consists of about 400 classes, 3,700 methods and a large test suite with more than 650 test cases. By this example, we will demonstrate test-driven fault navigation in the following Sections.

2.1 Typing Error Example in Seaside

We have inserted a defect into Seaside's Web server and its request/response processing logic (`WABufferedResponse` class, `writeHeadersOn:` method). Figure 1 illustrates the typing error inside the header creation of buffered responses. The typo in "Content-Lenght" is inconspicuous but leads to invalid results in requests that demand buffered responses. Streamed responses are not influenced and still work correctly.

Although the typo is simple to characterize, observing it can be laborious. First, some clients hide the failure since they are able to handle corrupted header information. Second, as the response header is built by concatenating strings, the compiler does not report an error. Third, by reading source code like a text, developers tend to overlook such small typos [28].

2.2 Challenges in Testing and Debugging

Localizing our typing error with standard tools such as test runner and symbolic debugger can be cumbersome. Figure 2 depicts a typical debugging session. First, Seaside's test suite answers with 9 failed and 53 passed test cases for all response tests. Since all failing runs are part of `WABufferedResponseTest`, developers might expect the cause within buffered responses. However, this

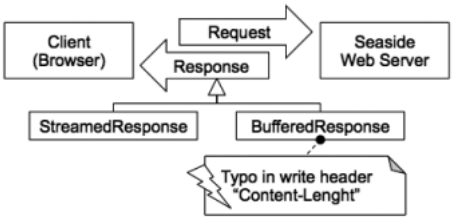

Fig. 1 An inconspicuous typo in writing buffered response headers leads to faulty results of several client requests.

assumption lacks evidence, such as a list of methods being executed by all failed tests. Second, starting the standard debugger on a failing test shows a violated assertion within the test method itself. This, however, means that developers only recognize the observable failure instead of its origin. Only the current stack is available, but our typo is far away from the observable malfunction. Third, the thrown assertion suggests that something is different from the expected response. Developers have to introspect the complete response object for localizing the typo. There are no pointers to the corrupted state or its infection chain. Remarkably, the response status is still valid (`200, OK`). Furthermore, in our example we assume that developers are aware of Seaside's request/response processing. However, developers' expertise significantly influences the required debugging effort; for instance, less experienced developers need more time for comprehending Seaside's continuation-based communication [27].

In general, debugging of test cases faces several challenges with respect to localizing failure causes and defects. On the one hand, testing only verifies if a failure occurs or not. There is no additional information about differences between failing and passing tests. Hence, developers can neither restrict the search space to particular program entities nor find expert knowledge for suspicious system parts. On the other hand, symbolic debuggers suffer from missing advice to causes and capabilities to follow the infection chain backwards. Since it is hard to understand how erroneous behavior comes to be, developers have to rely primarily on their intuition.

To support a breadth first search for debugging,

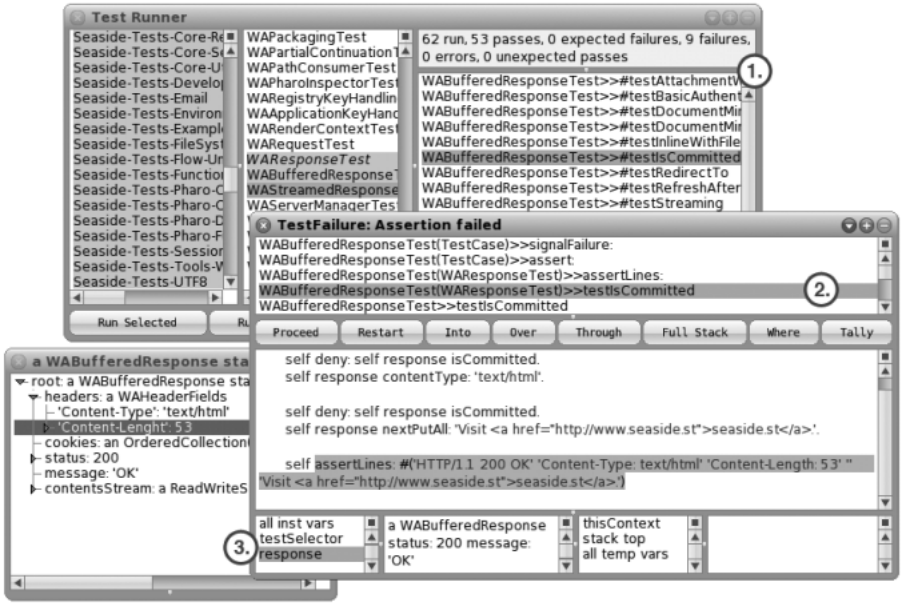

Fig. 2 Localizing failure causes with standard tools (Unit test runner (1), symbolic debugger (2), and object explorer (3)) is cumbersome.

spectrum-based fault localization techniques [1] compare the differences of test case coverage and restrict the potential search space. These techniques reveal anomalies in tested program behavior that could be responsible for failure causes. In Figure 3, a small example analyzes the number of failing and passing tests per covered method, computes a percentage value determining the failure cause probability, and returns a set of suspicious source code entities.

Although anomalies restrict the search space by providing excellent hints for starting debugging—there are often numerous anomalies that include neither a failure cause nor a defect. The amount of anomalies can be quite large because spectrum-based fault localization techniques suffer from a scalability problem [10]. Either they provide anomalies at the statement-level, whose dynamic analysis is slow and creates numerous results, or they include other program entities, such as methods or classes, which miss important fine-granular information. In addition to it, spectrum-based anomalies are only sorted by suspiciousness and the source code structure. Existing techniques do not consider the infection chain at all and so developers have to costly debug unrelated anomalies

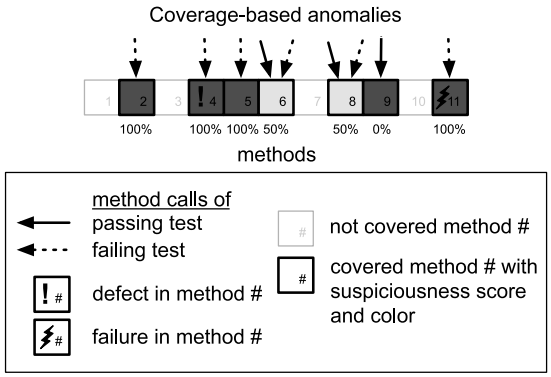

Fig. 3 Spectrum-based fault localization reveals a set of suspicious source code entities that could be responsible for failure causes. The static structure shows eleven methods (represented as boxes) in their source code definition order. Each covered method box is highlighted with its failure cause probability from red (high) to green (low).

one by one. In our small example in Figure 3, there are four methods with the same high suspiciousness value (red color) and it is not clear how they are related to failure causes, to each other,

and which statements are responsible for the failing behavior. For instance, it is not clear that method 2 is executed before the defect (method 4) and thus it works as expected (compare to Figure 4). As defects do not often come first, analyzing spectrum-based anomalies can be time-consuming and strongly depends on program comprehension.

For debugging failure causes or interpreting the list of suspicious entities, developers' expertise significantly influences the required effort [2]. More experienced developers invent better hypotheses about failure causes than novices that do not know the code base. Unfortunately, the identification of corresponding experts is quite challenging since observable failures do not explicitly reveal infected system parts. Existing approaches consider either the entire system or similar failure reports to automatically assigning bug reports to more experienced developers. Without taking suspicious system parts into account, these approaches consider a too large search space or they require a comprehensive bug tracker database with already fixed failures. For that reason, recommended developers are seldom good matches for debugging specific failures.

To understand how the failure comes to be, qualified developers have to follow the infection chain backwards [34]. Beginning with failure-reproducing behavior, in the form of failing test cases, developers trace observable failures or anomalies via the infection chain back to responsible defects. For instance, the small example in Figure 4 illustrates the infection chain with the observable failure (method 11, bottom right corner) and the defect (method 4, center left). For localizing the initial failure cause, developers have to decide at each executed method what the corrupted state or behavior is so that they are able to follow the infection chain backwards.

However, starting debugging at failures or anomalies still includes a long way to failure-inducing causes that happened in the past [15]. Most debuggers do not support back-in-time capabilities, and if they do, these features often come with a performance overhead [14] or a more complicated setup [26]. Furthermore, there is no direct navigation to failure causes and developers have to examine an enormous amount of data manually. The missing classification of suspicious or harmless behavior leads to numerous and often laborious decisions on which execution subtree to follow. Thus, the infection chain is hard to follow, developers require much program comprehension, and debugging of failing test cases becomes a time-consuming activity.

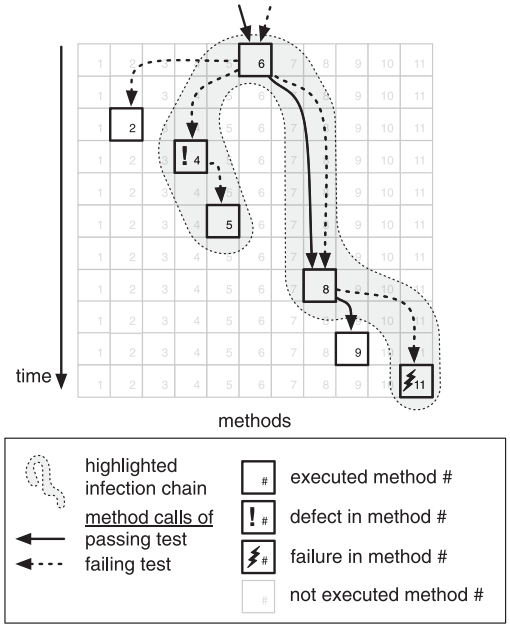

Fig. 4 Developers have to follow the infection chain (grey border) from the observable failure (method 11, bottom right corner) back to the defect (method 4, center left). Each row shows all eleven methods and highlights the specific method that is executed at this point in time. The different arrows define the method execution order for a passing or failing test case.

3 Test-driven Fault Navigation

We present test-driven fault navigation for improving debugging of failures that are reproducible by test cases. To address the challenges in testing and debugging, we integrate anomalies and failure causes and introduce a novel systematic top-down debugging process that guides developers with interconnected advice to failure causes.

3.1 Integration of Anomalies and Failure Causes

To further support debugging of failing test cases, we integrate anomalies and failure causes. Efficient debugging requires linked views between suspicious source code entities and erroneous behavior, as well as qualified developers for analyzing these entities. We expect that the combination of unit testing, spectrum-based anomalies, and lightweight back in time debugging is able to limit their shortcomings.

With the help of anomalies, we restrict the search space for failure causes in structure, the development team, and behavior of programs. In structure, we support developers in creating first hypotheses about failure causes by easily analyzing similarities of anomalies. To solve the scalability problem of existing spectrum-based fault localization, we split the coverage analysis of test cases over multiple runs. We provide fast access to method information and on-demand refinements at statements by re-executing test cases. In the development team, we assess expert knowledge for failures even though defects are still unknown. We only consider authors of spectrum-based anomalies and identify developers that understand suspicious system parts best. In behavior, we allow developers to easily navigate the infection chain backwards by highlighting potentially erroneous behavior. To get fast access to execution histories, we leverage reproducible test cases [32] and distribute their dynamic analysis depending on developers' needs [23]. By classifying traced methods with their spectrum-based results, developers are able to find abbreviations within the large amount of data.

Figure 5 illustrates the integration of anomalies and failure causes. First, there are three red anomalies in the structure (methods 5, 4, and 2 on the left side) that form a suspicious system part. Second, we expect that the most active developers of these three methods are experts for understanding this failure and its causes. Third, the infection chain is classified with anomalies and developers can directly start debugging on the left sub tree. For instance, methods 8 and 6 are less suspicious and developers can shorten the infection chain to methods 5, 4, and 2. Further on, as method 2 is called before the defect in method 4, developers, following erroneous behavior backwards, do not need to check this anomaly because they have already found the root cause.

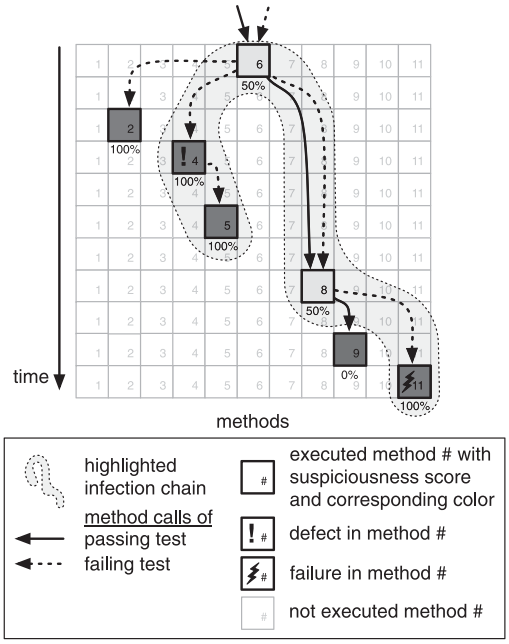

Fig. 5 The integration of anomalies and failure causes gives developers helpful advice on how to follow the infection chain backwards. The execution history includes information about failure cause probabilities at each single method. Thus, developers can decide where to focus the debugging effort and they understand how anomalies are related to each other.

3.2 Debugging Reproducible Failures

Besides the pure integration of anomalies and failure causes, localizing non-trivial faults also requires a systematic procedure [34]. Experienced developers apply a promising debugging method by starting with a breadth-first search [33]. They look at a system view of the problem area, classify suspicious system parts, and refine their understanding step by step. However, independent and specialized debugging tools does not coherently support such a systematic procedure. This often leads to confusing and time-consuming debugging sessions, especially for novice developers who trust more in intuition instead of searching failure causes systematically.

We introduce a systematic top-down debugging process with corresponding new tools that not only

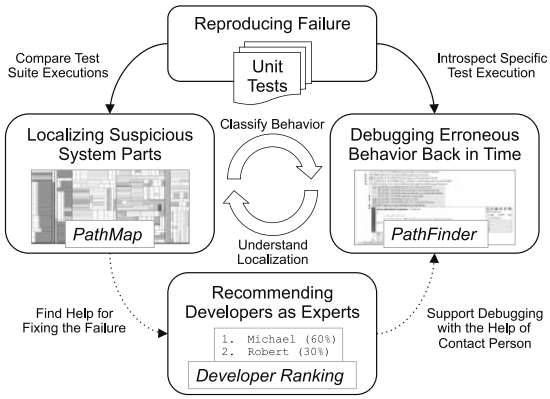

Fig. 6 Our debugging process guides with interconnected advice to reproducible failure causes in structure, behavior, and to corresponding experts.

supports the method of experts but also integrates anomalies and so provides guidances for all kinds of developers. Developers are able to navigate from failures to causes by reproducing observable faults with the help of test cases. Afterwards, they can isolate anomalies within parts of the system, identify other developers for help, and understand erroneous behavior. Figure 6 summarizes our test-driven fault navigation process and its primary activities:

Reproducing failure

As a precondition for all following activities, developers have to reproduce the observable failure in the form of at least one unit test. Besides the beneficial verification of resolved failures, we require tests above all as entry points for analyzing erroneous behavior. We have chosen unit test frameworks because of their importance in current development projects. Our approach is neither limited to unit testing nor does it require minimal test cases as proposed by some guidelines [3]. In the case of our Seaside example, developers have to implement a simple server request waiting for a corrupted response that cannot be parsed correctly.

Localizing suspicious system parts (Structural navigation)

Having at least one failing test, developers can compare its execution with other test cases and identify structural problem areas. By analyzing failed and passed test behavior, possible failure causes are automatically localized within a few suspicious methods so that the necessary search space is significantly reduced. For supporting spectrum-based fault localization within the system's structure, we have developed an extended test runner called *PathMap* that provides a scalable tree map visualization and a low overhead analysis framework that computes anomalies at methods and refines results at statements on demand. In Seaside, all failing tests overlap within response handling classes and the failure cause of our typing error can be isolated within a few methods.

Recommending developers as experts (Team navigation)

Some failures require expert knowledge of others so that developers can understand and debug faults more easily. By combining localized problem areas with source code management information, we provide a novel *developer ranking metric* that identifies the most qualified experts for fixing a failure. Developers having changed the most suspicious methods are more likely to be experts than authors of non-infected system parts. We have integrated our metric within PathMap providing navigation to suitable team members. For instance, our developer ranking metric proposes contact persons who have recently worked on the most suspicious buffered response methods.

Debugging erroneous behavior back in time (Behavioral navigation)

For refining their understanding of erroneous behavior, developers explore the execution and state history of a specific test. To follow the infection chain back to the failure cause, they can start our lightweight back in time debugger, called *PathFinder*, either at the failing test directly or at arbitrary methods as recommended by PathMap. If suspicious system parts are available, conspicuous methods classify the executed trace and so ease the behavioral navigation to defects. In our example, developers examine the request-response processing of a failing test in detail. Due to a classified trace, they can shorten the search for corrupted behavior to the creation of buffered response objects.

Besides our systematic process for debugging reproducible failures, the combination of unit testing and spectrum-based fault localization also provides the foundation for interconnected navigation with a high degree of automation. All activities and their anomalous results are affiliated with each other and

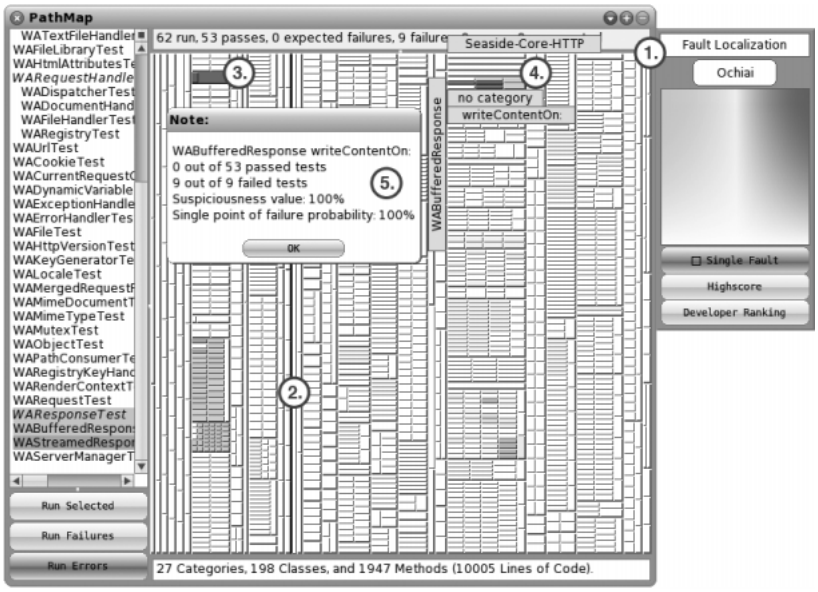

Fig. 7 PathMap is our extended test runner that analyzes test case behavior and visualizes suspicious methods of the system under observation.

so allow developers to explore failure causes from combined perspectives. Our tools support these points of view in a practical and scalable manner.

4 Structural Navigation: Localizing Suspicious System Parts

For supporting a breadth-first search, we provide a complete system overview that highlights anomalous areas for potential failure causes. Applying spectrum-based fault localization, which predicts failure causes by the ratio of failed and passed tests at covered methods, we analyze overlapping test behavior, identify suspicious system parts, and visualize the results. Our *PathMap* tool implements this approach as an extended test runner for the Squeak/Smalltalk development environment (Figure 7). Its integral components are a compact visualization in form of an interactive tree map, a lightweight dynamic analysis framework for recording test executions, and different fault localization metrics for identifying suspicious methods.

We visualize a structural system overview and its relation to test case execution in form of a compact and scalable tree map [31]. We reflect selected categories[†2] as full columns that include their classes as rows which in turn include methods[†3] as small boxes. The allocated space is proportional to the number of methods per node. All elements are organized alphabetically, and for a clear separation we distinguish between test classes on the left-hand side and core classes on the right-hand side (label 2 in Figure 7). The entire map can interactively be explored to get more details about a specific node (label 4 in Figure 7). Furthermore, each method can be colorized with a hue element between green and red for reflecting its suspiciousness score and a saturation element for its confidence. For instance, methods with a high failure cause probability possess a full red color. Such a visualization allows for a high information density at a minimal required space. The tree map in Figure 7 consists of only 500×500 pixels but is able to scale up to 4,000 methods. Even though this should suffice for most medium-sized applications, PathMap allows for filtering specific methods such as accessors, summarizing large elements, and resizing the entire tree map.

We ensure scalability of spectrum-based fault localization by efficiently recording test coverage with

†2 Categories are similar to packages in other programming languages.

†3 We provide Smalltalk's method categories as an optional layer, too.

method wrappers [4] and refining statement coverage on demand. To reduce the overhead of run-time observation, we restrict instrumentation to relevant system parts and dynamic analysis to the granularity level of methods. With the focus on selected categories we filter irrelevant code such as libraries where the defect is scarcely to be expected. For identifying failure causes in full detail, PathMap allows for refining coverage information inside specific methods. If developers request additional information for a method, we run all its covering tests, simulate byte code execution, and obtain required coverage information. We restrict the performance decrease of statement-level analysis only to the method of interest. Thus, we offer developers fast access to erroneous method behavior and optionally refinements for all details of suspicious statements.

Based on collected test coverage, we automatically identify anomalies and visualize suspicious methods in our tree map[†4]. In spectrum-based fault localization [11], failure cause probabilities are estimated by the ratio of all failing tests to test results per covered source code entity. Thus, methods are more likely to include the defect if they are executed by a high number of failing and a low number of passing tests. We distinguish between suspiciousness and confidence values of methods. While the former scores the failure cause probability with respect to covered tests and their results, the latter measures the degree of significance based on the number of all test cases. A lot of metrics for spectrum-based fault localization have been proposed among which Ochiai has shown to be the most effective one [1].

$$suspicious(m) = \frac{failed(m)}{\sqrt{totalFailed * (failed(m) + passed(m))}}$$

This formula returns a value between 0 and 1 for each method m being covered by at least one test. To visualize this result, we colorize method nodes in our tree map with a hue value between green and red. For instance, a suspiciousness score of 0.7 creates an orange area.

To assess the significance of a suspiciousness value, we apply a slightly adapted confidence metric. We only consider the relation between failed tests per method and all failing tests as we are not interested in sane behavior for fault localization.

$$confidence(m) = \frac{failed(m)}{totalFailed}$$

The returned value is directly mapped to the saturation component of already colorized method nodes. By looking only at faulty entities, we reduce the visual clutter of too many colors and results. For instance, a method covered by three out of six failing tests is grayed out.

Adapting spectrum-based fault localization to unit testing limits the influence of multiple faults. The effectiveness of existing spectrum-based approaches suffers from overlapping test cases describing different failures as well as coincidentally correct test cases which execute defects but do not verify their appearance. The selection of suitable unit test suites allows for ignoring such problematic tests and to focus on a single point of failure. Furthermore, based on the origination condition of single faults [30], which means each failure must evaluate the defect, PathMap optionally filters methods which were not executed by all failing tests. Thus, developers choose designated test suites, further reduce fault localization results, and concentrate on one specific failure at a time.

In our motivating typing error, PathMap localizes the failure cause within a few methods of Seaside's response classes. In Figure 7, developers only execute the response test suites as in ordinary test runners with the result of 53 passed and 9 failed tests (1). In the middle (2) they see a tree map of Seaside's structure with test classes on the left side and core classes on the right side[†5]. Each color represents the suspiciousness score of a method revealing anomalous areas of the system. For instance, the interactively explorable red box (3) illustrates that all nine failing tests are part of the buffered test suite. In contrast, the green box below includes the passed streaming tests and in orange shared test methods. The more important information for localizing the failure cause is visualized

†4 For statements, we compute spectrum-based anomalies with the same formulas as for methods and show their results in Smalltalk's source code browser directly.

†5 For the purpose of clarity, we limit the partial trace to Seaside's core.

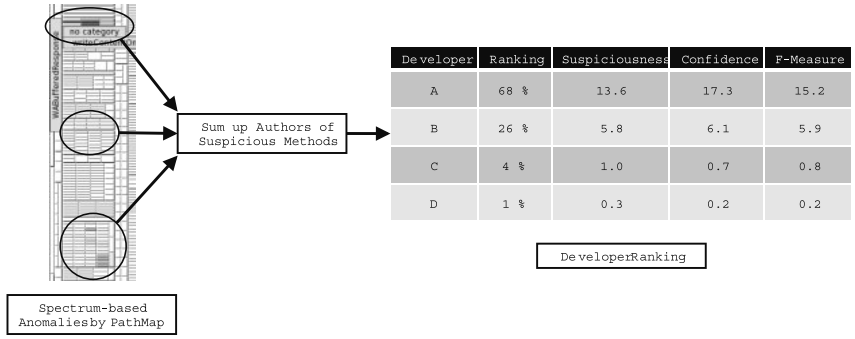

Fig. 8 Our developer ranking points out (anonymized) experts. Based on authors of spectrum-based anomalies, we create a ranked list of possible experts that understand failure causes best.

at (4). There are three red and orange methods providing confidence that the failure is included in the `WABufferedResponse` class. To that effect, the search space is reduced to six methods. However, a detailed investigation of the `writeContentOn:` method (5) shows that it shares the same characteristics as our failure cause, i. e., `writeHeadersOn:`. At this point, it is not clear from a static point of view how these suspicious methods are related to each other.

5 Team Navigation: Recommending Developers as Experts

As understanding anomalies and failure causes requires thorough familiarity with suspicious system parts, we propose a new metric for identifying expert knowledge. Especially in large projects where not everyone knows everything, an important task is to find experts that are able to explain erroneous behavior or even fix the failure itself [2]. Assuming that the author of the still unknown defect is the most qualified contact person, we restrict the search space to suspicious system parts and approximate developers that have recently worked on corresponding methods. Consequently, as anomalies have a high probability to include failure causes [11], our anomaly-based metric recommends developers that comprehensively understand infections or possibly the defect itself. Figure 8 summarizes our metric and its relationship to PathMap's anomalies. For calculating the developer ranking, we sum up suspicious and confident methods for each developer, compute the harmonic mean for preventing outliers, and constitute the proportion to all suspicious system parts.

First, from all methods of our system under observation ($M_{Partial}$) we create a new set that includes methods being identified by the spectrum-based fault localization.

$$M_{Suspicious} = \{m \in M_{Partial} \mid suspicious(m) > 0\}$$

Second, with the help of Smalltalk's source code management system we identify developers that have implemented at least one of these suspicious methods. Having this list, we divide suspicious methods into one set per developer based on the method's most active author. The function *authorOf()* is independent of our approach and can be replaced by arbitrary heuristics that return expert knowledge for a specific method such as most activity, last access, and initial implementation.

$$M_{Developer} = \{m \in M_{Suspicious} \mid authorOf(m) = Developer\}$$

Third, for a specified set of methods we sum up suspiciousness and confidence scores and create a weighted average of both. The harmonic mean combines both values and prevents outliers such as high suspiciousness but low confidence.

$$FScore(M) = 2 \cdot \frac{\left(\sum_{m \in M} suspicious(m)\right) \cdot \left(\sum_{m \in M} confidence(m)\right)}{\sum_{m \in M} suspicious(m) + confidence(m)}$$

Fourth, we normalize individual developer scores by comparing them with the value of all suspicious methods.

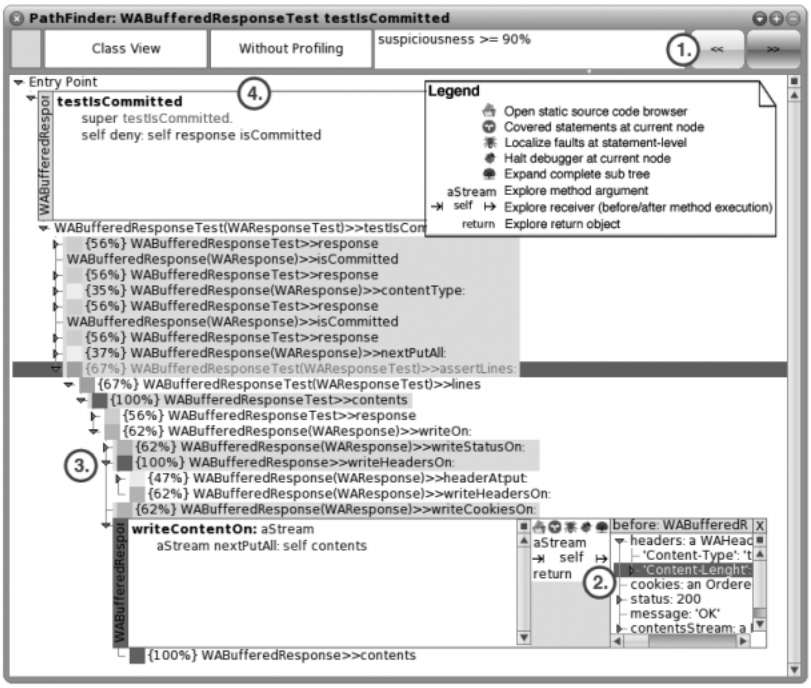

Fig. 9 PathFinder is our lightweight back in time debugger that classifies failing test behavior for supporting developers in navigating to failure causes.

$$developerRanking(Developer) = \frac{FScore(M_{Developer})}{FScore(M_{Suspicious})}$$

Finally, we sort all developers by their achieved expert knowledge for the anomalous system parts so that we estimate the most qualified contact persons even though the defect is not yet known.

With respect to our typing error, we reduce the number of potential contact persons to 4 out of 24 Seaside developers, whereby the author of the failure-inducing method is marked as particularly important. The table in Figure 8 summarizes the (interim) results of our developer ranking metric and suggests Developer A for fixing the defect by a wide margin. With our fault-based team navigation, we do not want to blame developers but rather we expect that the individual skills of experts help in comprehending and fixing failure causes more easily.

6 Behavioral Navigation: Debugging Erroneous Behavior Back in Time

To follow corrupted state and behavior back to failure-inducing origins, we offer fast access to failing tests and their erroneous run-time data. Based on our previous work [23], we extend our *PathFinder* tool to a back in time debugger for introspecting specific test executions with a special focus on fault localization. It does not only provide immediate access to run-time information, but also classifies traces with suspicious methods. For localizing faults in test case behavior, developers start exploration either directly or out of covered suspicious methods as provided by PathMap. Subsequently, PathFinder opens at the chosen method as shown in Figure 9 and allows for following the infection chain back to the failure cause. We provide arbitrary navigation through method call trees and their state spaces. Besides common back in time features such as a query engine for getting a deeper understanding of what happened, our PathFinder possesses two distinguishing characteristics. First, step-wise run-time analysis allows for imme-

diate access to run-time information of test cases. Second, the classification of suspicious trace data facilitates navigation in large traces.

We ensure a feeling of immediacy when exploring behavior by splitting run-time analysis of test cases over multiple runs [23]. Usually, developers comprehend program behavior by starting with an initial overview of all run-time information and continuing with inspecting details. This systematic method guides our approach to dynamic analysis: run-time data is captured when needed. *Step-wise run-time analysis* consists of a first shallow analysis that represents an overview of a test run (a pure method call tree) and additional refinement analysis runs that record on-demand user-relevant details (e.g. state of variables, statement coverage for spectrum-based fault localization). Thereby, unit test cases fulfill the requirement to reproduce arbitrary points on a program execution in a short time [32]. Thus, by dividing dynamic analysis costs across multiple test runs, we ensure quick access to relevant run-time information without collecting needless data up front.

We classify behavior with respect to suspiciousness scores of methods for an efficient navigation to failure causes in large traces. Therefore, we either reuse PathMap's already ranked methods or rerun the spectrum-based fault localization on traced methods again. The trace is divided into more or less erroneous behavior depending on test results of called methods. On the analogy of PathMap, we colorize the trace with suspiciousness and confidence scores at each executed method. Moreover, a query mechanism supports the navigation to erroneous behavior. We expect that our classified traces identify failure causes more quickly as it allows shortcuts to methods that are likely to include the defect.

In our Seaside example, PathFinder highlights the erroneous behavior of creating buffered responses and supports developers in understanding how suspicious methods belong together. Following Figure 9, developers focus on the failing `testIsCommitted` behavior. They begin with the search for executed methods with a failure cause probability larger than 90 % (1). The trace includes and highlights four methods matching this query. Since the `writeContentOn:` method (2) has been executed shortly before the failure occurred, it should be favored for exploring corrupted state and behavior first[†6]. A detailed inspection of the receiver object reveals that the typo already exists before executing this method. Following the infection chain backwards, more than three methods can be neglected before the next suspicious method is found (3). Considering `writeHeadersOn:` in the same way manifests the failure cause. If necessary, developers are able to refine fault localization at the statement-level analogous to PathMap and see that only the first line of the test case is always executed, thus triggering the fault (4).

7 The Paths Framework

In this section, we outline the implementation of test-driven fault navigation with the help of our Paths framework. After that, we discuss our approach with respect to other programming languages and its introduction to existing software systems.

7.1 Implementation

Test-driven fault navigation is based on our Paths framework that is illustrated in Figure 10. It is seamlessly integrated into the *Squeak/Smalltalk* development environment and requires only source code, unit tests, and a corresponding change history of a selected Smalltalk project. In Squeak, the latter is automatically recorded by a built-in source code management system. After each change, author credential, timestamp, and modifications are stored in a new version of the related source code artifact.

The *Paths dynamic analysis framework* supports the observation of unit test behavior by implementing the refined coverage and step-wise run-time analysis. Refined coverage analysis [22] is a lightweight approach that splits dynamic analysis and its costs over multiple test runs. First, it rapidly records the relationship between unit tests and executed methods. Second, the statement coverage of selected methods can be refined on demand by re-executing their corresponding test cases. With the help of this analysis, PathMap reveals anomalies for our structural navigation within a short amount of time and at different levels of

†6 The simple accessor method `contents` can be neglected at this point.

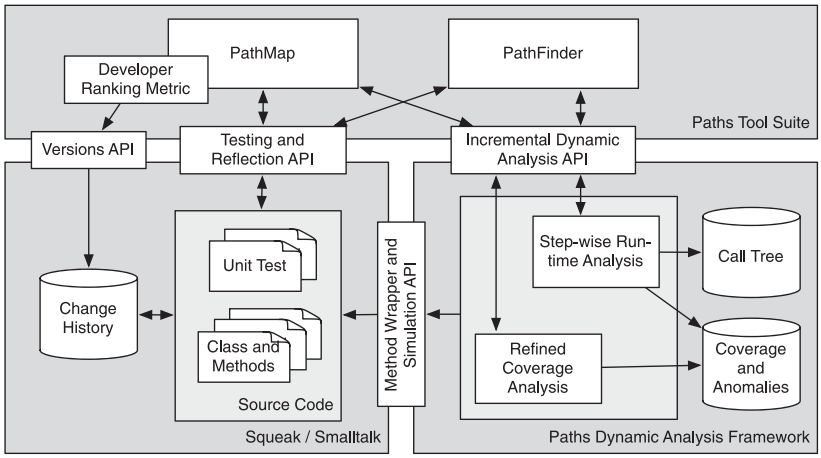

Fig. 10 The Paths framework is integrated into the Squeak/Smalltalk development environment and consists of a dynamic analysis framework and our tool suite.

detail. Step-wise run-time analysis [23] divides the dynamic analysis of one specific test case over multiple runs. In a first shallow analysis, it records a simple method call tree which is presented in PathFinder as our behavioral navigation. If desired, additional information such as states can be collected by running and analyzing the same test again. For recording run-time information, we rely on method wrappers and Smalltalk's interpreter simulation. At the level of methods, we collect run-time information with flexible method wrappers [4]: a wrapper introduces new behavior before and after the execution of a specific method without changing its behavior. Depending on the chosen analysis technique, wrappers collect among others coverage, method calls, and state refinements. To record statements of a specific method, a special wrapper starts and stops Smalltalk's simulation engine that analyze dedicated byte codes only. Both analysis techniques are necessary since a full simulation would slow down the execution by a factor of at least 100 [8]. Finally, the framework stores all collected measurements and makes this data available to any interested tool. For instance, PathFinder can reuse coverage information for highlighting anomalies in test case behavior.

Our *Paths tool suite* consists of the extended test runner PathMap, the developer ranking metric, and the back in time debugger PathFinder. PathMap requires Smalltalk's testing API to control the underlying unit test framework, the reflection API to draw a tree map of the system structure, and the incremental dynamic analysis API to reveal anomalies with the refined coverage analysis. Our developer ranking metric is embedded into PathMap and combines its anomalies with author information of the change history. PathFinder applies the testing and reflection API to control specific test runs and to show source code in corresponding call trees. Such call trees are built with the help of our incremental dynamic analysis API that starts step-wise run-time analysis and assigns anomalies.

7.2 Discussion

We argue that our approach can easily be adapted to other object-oriented programming languages that include a unit test framework. For implementing our Paths framework, the language and its libraries have to support dynamic and static analysis techniques. While the dynamic analysis of method executions can be implemented with aspect-oriented programming [7], statement-level coverage depends on language features. For instance, in C++ many coverage tools insert probes into the source code and in Python the interpreter offers a simple hook function for a fine-grained run-time analysis. Regarding static analysis, developers can rely on several external analysis tools or the reflection capabilities of their language. Also, many version control systems such as subversion offer interfaces to request author information of previous changes. Finally, our Paths tool suite is mostly

a visualization concept whose implementation only depends on the underlying IDE user interface. For instance, Eclipse can be extended with a plug-in for rendering the tree map and suspicious data.

The introduction costs for test-driven fault navigation are low in the beginning but it will take some time to become well acquainted with our tools. These costs are especially composed of adapting the underlying software system, teaching our new debugging process, and practicing with our Paths tools. The preparation effort for the underlying software system is negligible. Once our Paths tool suite is available for a specific programming language, it only requires access to source code, unit tests, and the change history. We have already analyzed numerous Smalltalk projects without any problems. Only, in some cases we had to revise non-deterministic unit tests or we could not analyze the system because of extensive reflection mechanisms. The new concepts of our test-driven fault navigation process are easy to understand since they are similar to other debugging activities. In comparison to the "traffic" principle [34], developers also start with a breadth first search and they follow the infection chain back. For instance, after presenting our approach in a 90 minutes lecture, our undergraduate students confirmed that they have understood our approach. They praised the possibilities to refine hypotheses with anomalies and to navigate the infection chain backwards. However, we also learned from our students that practice with our Paths tool suite needs some time. Although they come acquainted with the main features in a few hours, an efficient debugging session still looks different. On the one hand, highlighted anomalies in PathMap and PathFinder as well as the developer ranking are easily applicable. On the other hand, debugging back in time and searching anomalies in large traces require an expensive change of thinking when localizing more complicated failure causes. For these reasons, we will prepare additional tutorials for getting started with the more advanced features of our tools.

8 Evaluation

We evaluate test-driven fault navigation with respect to its practicability for developers, the accuracy of our developer ranking metric, and the efficiency of our Path tool suite.

8.1 Practicability of Test-Driven Fault Navigation

For evaluating test-driven fault navigation and its corresponding tools, we conduct a user study within the scope of a real world project. We observe six developers while debugging six failures and compare PathMap and PathFinder with Smalltalk's test runner and symbolic debugger. In summary, we find out that test-driven fault navigation is able to decrease debugging costs with respect to required time and developer's effort.

Experimental Setup

We choose Squeak's *iCalendar* project[†7] as the underlying software system for our user study. iCalendar is a library that supports the identically named file format for sharing meeting requests and tasks independent of specific calendar applications. The project implements import and export functionality of the file format including a parser, a domain-specific object model, and I/O handling. It is an external, open source, and real world project that is used in several other applications. We choose iCalendar because of its maturity, already included and comprehensive test base, understandable domain, and ideal size that is neither too small nor too large. The project characteristics can be seen in the upper part of Table 2.

For our user study, we observe six developers with a similar background knowledge. The participants are computer science students in the 5th semester. All of them have between 3 and 11 years of programming experience and professional expertise with symbolic debuggers. In the last year, they attended two of our software engineering courses, in which we intensively taught object-oriented programming with the help of Smalltalk. In this time, they built a new system from scratch, maintained an existing application, and passed the courses with excellent grades. The chosen students have similar development skills and they become acquainted with iCalendar in our user study for the first time. Thus, we can ensure that the required debugging effort is not much influenced by individual skills and knowledge about the system.

During the study, these students are supposed

†7 http://www.squeaksource.com/ical/

Table 1 Description of iCalendar's failures.

Failure	Description	Difficulty	call nodes between failure and defect	Suspicious method rank
1	Reversed comparison operator of event objects	Easy	1	1
2	Wrong conditional statement in handling address parameters	Easy	27	22
3	Unintended string constant in phone types	Normal	5	68
4	Forgotten deletion of obsolete calendar events	Normal	84	10
5	Missing separator for parsing event files	Hard	520	31
6	Return of improper but polymorph objects for storing alarms	Hard	2133	42

to localize six failures with different levels of difficulty, which are described in Table 1. We insert these six defects all over the iCalendar system, whereby we described them obviously. For instance, we comment important statements instead of deleting them. So, it is easier for our developers to verify defects after they have followed the infection chain backwards. For each failure, we assess a level of difficulty that estimates the required effort to localize its defect. On the one hand we choose this level depending on our own debugging experience on the other hand it also reflects the length of the infection chain (call nodes between failure and defect) and the position of the defect in the list of anomalies (suspicious method rank). Finally, we have two failures from each of six, which are easy, normal, and hard to debug. All failures can be reproduced with 1-10 failing test cases, which do not trivially include defects as part of their stack traces.

User Study Procedure

Before our user study, we presented a software engineering course, in which all students had already access to our Path tools for a period of three months. In this course, we also taught them advanced debugging concepts [34] and we introduced test-driven fault navigation as an exemplary approach. To find participants for our user study, we performed a short questionnaire with all students, which takes into account years of programming experience, reviews of course projects, and grades. Based on these results, we selected six excellent students with a similar background knowledge. However, the questionnaire also revealed that mostly all students were afraid of doing test-driven fault navigation because of the new debugging concept and neglected guidance from our part.

For the preparation of our participants, we introduced test-driven fault navigation again and we allowed them to become acquainted with iCalendar. Within two hours, we first presented our Seaside typo error followed by an instructed and comprehensive practice with our tools. The students debugged one example failure in iCalendar with the help of our guidance. In doing so, they understood iCalendar's basic concepts, investigated its source code, and learned to debug with test-driven fault navigation.

We conduct the user study by observing our developers during debugging iCalendar's failures with different tools. First, all developers debug three failures with Squeak's standard debugging tools. After that, they debug the remaining three failures with our Path tools. While using test-driven fault navigation, we provide assistance with handling PathFinder's user interface and its features. We do not influence the process of localizing failures. For all six failures, we measure the complete debugging time, including everything from time to consider source code to execution time of applied tools. Additionally, for test-driven fault navigation we notice the point in time when our developers start PathFinder to follow the infection chain back. If the defect is not localized after 15 minutes, we mark the failure as not solved.

To evaluate our approach, we assign each developer three failures for debugging with standard tools (symbolic debugger and test runner) and three failures for test-driven fault navigation (Path-

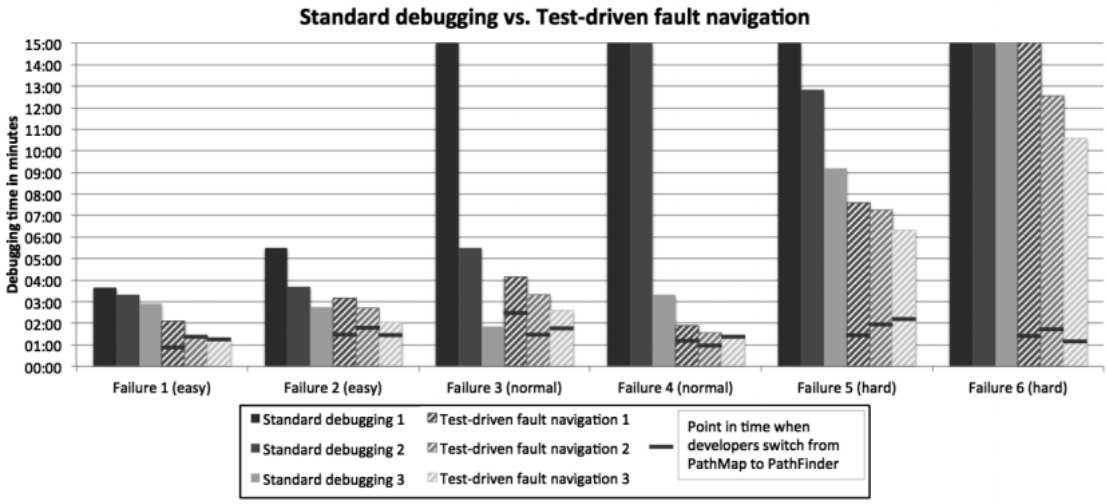

Fig. 11 Required debugging time with symbolic debugger and test runner compared to PathFinder and PathMap. The time includes all applied debugging activities such as running tests, time to consider source code, and execution time of tools. Lines in test-driven fault navigation results mark the point in time when developers started PathFinder to follow the infection chain back.

Finder and PathMap). For each of the six failures, we ensure a unique combination of developers and applied tools. At each level of difficulty a developer debugs one failure with standard tools and the other one with our Path tools.

After the study, we interviewed each student and asked for "I like and I wish" feedback with respect to our approach and Path tools.

Lessons Learned

With the help of our user study, we reveal that test-driven fault navigation is able to reduce the required effort for localizing failure causes. Compared to standard debugging tools, developers, which apply our Path tools, need in the majority of cases less time for debugging. Figure 11 summarizes for each failure the required debugging time with standard tools and our Path tools.

In the case of the first two easy failures, developers with test-driven fault navigation are about one minute faster compared to developers with symbolic debuggers and test runners only. For instance, while the first failure requires at least three minutes to debug with standard tools, our Path tools are able to localize the defect in less than two minutes. Even if all developers exchange their tools for the second failure—from standard tools to Path tools and vice versa—test-driven fault navigation is almost always faster. Thus, our performance increase is independent of the expert knowledge of individual developers.

We have similar results with the next two normal failures but the differences in required debugging time are considerably larger. Especially, developers with standard tools have some problems in localizing failure causes. Three students could not find the defect within 15 minutes because they did not comprehend what is going wrong. In contrast, two other developers required only a short time for debugging with standard tools. While one developer knows the infected source code very well from the preparation phase, the other developer instantly had a proper intuition. Nevertheless, test-driven fault navigation is once more better and allows all developers to localize failure three in less than four minutes and failure four in less than two minutes. The integration of anomalies and failure causes is very helpful and restricts the search space a lot. With the help of PathFinder, developers could easily understand how the failure comes to be and they were able to directly jump into erroneous behavior.

The last two hard failures required with all tools much more time and they could not be solved by

five developers including one participant with Path tools. We argue that these two failures are very hard to debug because the corresponding test cases fail far away from the failure-inducing defect. In the case of failure five, all test-driven fault navigation participants were able to identify the defect within six to eight minutes, while standard debugging tools either could not solve this failure or need still more time. Failure six was so difficult that no developer could solve it with standard tools. Also with our Path tools one student could not solve it—although he was close by. The other two developers were able to find this failure after about twelve minutes. Especially, PathFinder's possibility to follow the infection chain backwards in combination with anomalies supported the developers in isolating failure causes.

During debugging, we observed the participants and got some interesting insights. Developers with standard tools rely primarily on their intuition. Often, they guessed what is going wrong and what is the infected state or behavior. In doing so, some developers had proper hypotheses about failure causes but no developer was constantly better in guessing than another one. This is also reflected in the differences of required debugging time. In contrast, our test-driven fault navigation allows developers to rely on a systematic debugging process and the advice of our tools. They linked the corresponding anomalies with their hypotheses and followed the infection chain backwards. All developers take advantage of the combined perspectives of our Path tools. They usually started with a breadth-first search in PathMap and then followed the infection chain through suspicious behavior. As a consequence, the differences in debugging time are often marginal.

For assessing the effectiveness of PathMap and PathFinder, we noticed the point in time when developers switched from localizing suspicious system parts to debugging erroneous behavior back in time. For each failure, we marked this point in time with small lines in the corresponding columns of Figure 11. Usually, PathMap has been applied for a breadth-first search in the first two minutes. During this time, developers got a first impression of anomalous system parts and they built first hypotheses about failure causes. In some cases (failure 1 and 4), these hypotheses were sufficient to localize defects without the help of PathFinder. In both failures, the defected method was ranked in a very suspicious anomaly (compare to Table 1) so that developers saw from the visualization alone where the defect might be located. However, these developers could not explain why such defects result in observable failures. If defects could not obviously be found with PathMap, developers started PathFinder and followed the infection chain back. The required debugging time ranges from one to twelve minutes and strongly depends on the difficulty of failures and at which anomalous method developers opened a failing test case. In summary, both tools are effective for debugging because they allow developers to reduce the search space step by step until they localize the root cause.

Feedback of Participants

After the user study, we interviewed the participants and asked them for "I like and I wish" feedback about our approach and tools. All developers like our Path tools and conceive them as very valuable for debugging. A participant summarizes it as follows: *"Especially while debugging non-trivial faults, I have the feeling that the failure localization requires less time. The classification of anomalies allowed me to invent better hypotheses about the failure cause and to abbreviate the execution history."* Another student stated: *"The tools are fast and very well integrated with each other. The coloring of map and trace has helped a lot in focusing on suspicious entities."* A third developer mentions: *"Compared to a standard debugger, I had no problem to see into a complete program execution and it was easy to understand how a failure comes to be."* Last but not least, one of them concluded *"I can very well imagine that the Path tools improve debugging of our real failures, too."*

Besides the positive feedback, they mostly wish usability improvements of our tools. They suggest promising new user interface elements and advanced query mechanisms for searching the call tree and its states. We will implement most of the recommendations in the near future. Another point is that test-driven fault navigation is a new debugging method which requires a change of thinking. They wish to have more practice in order to debug with our Path tools even better. For this reason, we will make our approach available within our next software engineering course. Moreover, we will extend

Table 2 Project statistics and average performance characteristics for PathMap, our developer ranking metric, and PathFinder

		Seaside	iCal-endar	4Confer-ences	AweSOM	Compiler
Project statistics	Classes	394	77	175	68	64
	Methods	3708	1347	2540	742	1294
	Tests	674	115	89	124	49
	Coverage	58.3 %	72.98 %	43.74 %	81.8 %	51.1 %
PathMap	Exec. time all tests (s)	9.19	1.05	198.57	3.77	0.91
	Δ Fault localization (s)	26.61	1.33	8.64	8.70	1.85
	Refined fault localization (s)	2.38	2.51	16.70	3.73	0.90
Devel. Ranking	Computation Costs (s)	11.19	5.00	7.92	3.44	4.49
PathFinder	Exec. time per test (ms)	0.76	3.34	1910.82	17.33	7.69
	Δ Shallow analysis (ms)	336.17	258.16	281.02	235.79	247.23
	Δ Refinement analysis (ms)	16.92	2.67	19.65	5.93	2.15

our upcoming debugging lecture with a test-driven fault navigation tutorial.

Finally, all developers confirmed that the combination of anomalies and failure causes is promising for debugging with less effort. With our test-driven fault navigation, they get helpful advice for strengthening their hypotheses about failure causes and it appeared easier for them to follow the infection chain backwards.

8.2 Accuracy of Recommended Developers

We evaluate our developer ranking metric with the help of a web-based conference management system. The conference management system called 4Conferences permits activities such as the registration of attendees, the organization of payments, the printing of badges, and the planning of talks. It has been developed by five undergraduate student projects in the context of two software engineering courses in the last two years. In these courses, the first three authors act as customers for the 28 involved students. The students implemented 4Conferences with the help of Extreme Programming. Due to agile practices such as collective code ownership, all developers had access to the entire code base and each method was implemented by a number of students. 4Conferences's project characteristics can be seen in the upper part of Table 2.

To measure the accuracy of our developer ranking metric, we introduce a considerable number of defects into 4Conference. We randomly choose 100 covered methods that are neither part of test code nor trivial (e. g., getters) and create for each method a defect with a small mutation engine. For instance, we hide the method body and returning the receiver object instead of executing the original functionality. We ensure the occurrence of each failure by observing corrupted test cases. For each of the 100 defects, we automatically run PathMap, compute a sorted list of recommended developers, and compare it with the *most active author* of the faulty method. In this evaluation, we consider such an author as the most qualified expert for understanding the specific defect.

Figure 12 shows the distribution of methods per author, where a method is always associated to its most active author. This distribution reveals that 4Conferences has been uniformly implemented by our 28 students. Only two developers have developed more than 10 % of the system, respectively 18 % and 11 %, and eleven developers have realized less than 2 % of all methods. If we randomly insert defects into 4Conferences, we have a chance of up to 18 % to pick the most qualified developer from the static distribution of expert knowledge. Furthermore, there is a maximum probability of 38 % to have the expert within the first three choices.

Figure 13 illustrates developer recommendations of a simple coverage metric that takes into account only covered methods of failing tests. This metric restricts the search space to erroneous related methods and sums up their authors. The chart shows the position of the most qualified developer

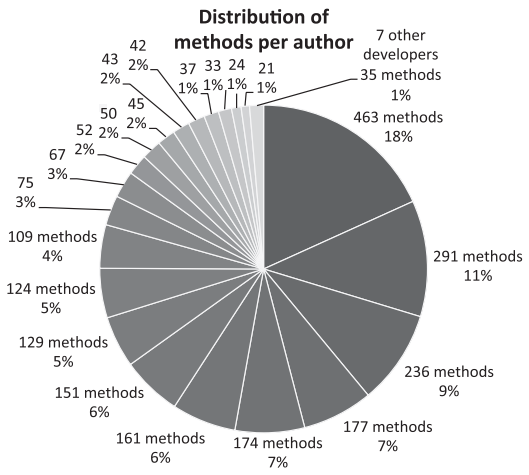

Fig. 12 Results for developer recommendation based on method distribution.

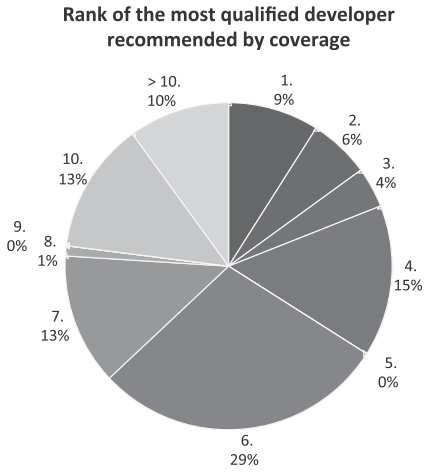

Fig. 13 Results for developer recommendation based on coverage data.

in the list of recommended experts. In less than 20 % of all defects, the expert is found within the first three ranks. Compared to the static distribution of methods per developer, this value is below the 38 % of choosing an arbitrary expert. Only if we consider at least six recommendations and ignore the five false suggested experts, this metric seems to be better.

Figure 14 presents for our anomaly-based developer ranking metric the position of the most qualified expert in the list of recommendations. For almost a third of all defects, the responsible developer of the faulty method is ranked in the first

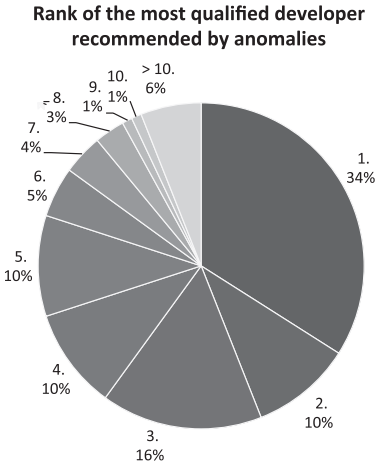

Fig. 14 Results for developer recommendation based on anomalies.

place. With a probability of 60 %, we suggest the expert within the first three ranks and with a probability of 80 % in the first five ranks. Since we restrict the search space to anomalies and their involved developers, our metric is able to outperform the both other approaches. For instance, we achieve a probability of 60 % followed by 38 % of developer distribution and 19 % of the coverage metric for recommending experts in the first three ranks. Even if developers being responsible for the fault are not listed at the top, we expect that their higher ranked colleagues are also familiar with suspiciously related system parts. Considering that failure causes are still unknown, our developer ranking metric achieves very satisfactory results with respect to the accuracy of recommended experts.

8. 3 Efficiency of the Path Tool Suite

We evaluate the low performance overhead of our Path tools by measuring the required time for collecting and presenting run-time information from five different Smalltalk projects.

The properties of the five mid-sized Smalltalk projects are summarized in the upper part of Table 2. Of the five projects, three systems are already described in this paper as we apply test-driven fault navigation to them. Seaside is an open source Web framework that includes our motivating typo error. iCalendar handles appointments and lays the foundation for our user study. 4Confer-

ences supports the web-based management of conferences and determines the accuracy of our developer ranking metric. All projects, including their test suites, were not implemented by the authors. The remaining two (AweSOM and Compiler) are additional projects that have also profited from our Path tools during their maintenance. AweSOM is a research prototype developed in our group that implements a virtual machine for SOM Smalltalk on top of the Squeak system. The last project is Squeak's standard Smalltalk compiler.

All five projects are well tested and in daily use in software development and business activities. Their acceptance, integration, and unit tests cover large parts of the system, imposing different computational costs. The projects involve various application domains such as end-user Web applications, development tools, and virtual machines. As a consequence, we argue that our Path tools are applicable to a broad range of different software systems if they include an adequate number of test cases.

We measure the run-time overhead of our tool suite by analyzing all tests of the five projects. We do not insert failures because the analysis overhead is independent of their occurrence. PathMap executes the entire test suite with and without fault localization. We refine fault localization at statements only at non-trivial methods including a McCabe complexity [18] greater than one. So, we exclude simple methods with sequential behavior where all statements have the same suspiciousness score. For our developer ranking metric, we assume the worst case by declaring each method as anomalous and computing the most active developer. Thus, our metric has to gather all method commits of a project. PathFinder runs each test on its own and analyzes the overhead produced by step-wise run-time analysis. Shallow analysis records the entire method call tree while refinement analysis creates a deep copy of the returned object of a random call node. All experiments were run on a MacBook with a 2.4 GHz Intel Core 2 Duo and 8 GB RAM running Mac OS X 10.6.6, using Squeak version 4.1 on a 4.2.1b1 virtual machine.

The average results for the performance of PathMap are described in the second part of Table 2. The first row shows the pure run-time (in seconds) for executing all tests. While four projects run their test suites in less than ten seconds, the 4Conferences project requires more than three minutes due to several slow running acceptance tests. These tests check with the help of Selenium[†8] complete user workflows including multiple interactions with the Web application. The second row presents the overhead resulting from spectrum-based fault localization. PathMap's fault localization slows down execution by a factor of 2.3 for iCalendar to 3.9 for Seaside. In the case of 4Conferences, long waiting times for responding to interactions of acceptance tests explain the minimal slow down factor of 1.04. In all other cases, the variation originates from additional instrumentation and visualization costs. Nevertheless, this overhead is low enough for applying spectrum-based fault localization frequently. The third row reveals time for refining fault localization at statements. The time mostly depends on the number of covering tests per method and their corresponding run-time. In most cases, developers get refined results for complex methods in less than three seconds (without 4Conferences's long running acceptance tests the 80th percentile is below 2.6 seconds). We argue this time is still acceptable compared to a complete statement coverage analysis that possesses a slow down factor of about 120 in Squeak/Smalltalk [8].

The third part of Table 2 presents the computation costs for our developer ranking metric. The required time for our metric is between 3.5 seconds for AweSOM as the smallest and about 11 seconds for Seaside as the largest system. The computation costs strongly depend on the number of analyzed methods. For each method, our metric requests Smalltalk's version control system to get all changes and to compute the most active author. Especially, the access to method changes slows down the execution as it includes excessive I/O handling. Considering our worst case scenario that each method is an anomaly, the mentioned computation costs reflect the maximum per project. In a realistic setting anomalies cover only a subset of all methods and so we argue that the computation costs are still acceptable. For instance, in our Seaside typing error example we return the result of 54 anomalous methods in about 3 seconds.

The lower part of Table 2 describes the average

†8 http://www.seleniumhq.org/

performance characteristics of PathFinder. The first row lists the average and pure run-time per unit test. As before 4Conferences's acceptance tests include high run-time costs which hinder immediate feedback by our tools. Apart from that, executing a single unit test requires less than 20 milliseconds on average. The second row demonstrates the overhead associated with building the lightweight method call tree. On average, this value is between 200 and 300 milliseconds meaning that the shallow analysis is quite fast. The 99th percentile for the required run-time overhead is below 750 ms. The third row deals with the refinement analysis that allows developers to reload state information on demand. In doing so, the required analysis overhead only depends on the effort for creating a deep copy. Thus, our refinement analysis also imposes minimal costs as the 95th percentile is below 25 ms for all test cases. In summary, stepwise run-time analysis supports fast response times when debugging a test execution back in time since run-time data is in most cases provided in considerably less than two seconds [23].

8.4 Threats to Validity

First, the Smalltalk context of our evaluation might impede validity by limited scalability and general applicability. However, the Seaside Web framework is in fact a real-world system and it exhibits source code characteristics comparable to particular complex Java systems such as JHotDraw [23]. Even if the remaining projects are only mid-sized systems, they illustrate the applicability of our Path tools once unit tests are available. While these insights do not guarantee scalability to arbitrary languages and systems, they provide a worthwhile direction for future studies assessing general applicability.

Second, our user study is only based on one particular project, its six failures, and undergraduate students. We consider the iCalendar project as a real world application since it is mature and an important part of several other systems. The six failures are realistic and based on similar known defects that we have found in other projects. We treat our students as professional developers because of their longstanding programming experience. Although we require a larger study for a general conclusion, our user study already reveals the benefits of our approach and its tool suite.

Third, the chronological order of debugging the first three failures with standard tools could positively influence participants' program comprehension. Thus, there is a chance to localize the remaining three failures with our Path tools more simply. To reduce this factor, we have a preparation phase of two hours to become acquainted with iCalendar. During this time, developers read not only source code but also applied PathFinder to understand behavioral examples of test cases [23]. Furthermore, we make sure that all defects and their infection chains are unique. They are located in completely different system parts and their failure-reproducing test cases do not overlap each other.

Fourth, we limit the evaluation of our developer ranking metric to accuracy and performance characteristics. We have not yet checked the effectiveness in a realistic setting due to difficulties in finding suitable Smalltalk projects. Many projects suffer from an unbalanced distribution of method authors, missing tests that reproduce failures, or failure reports that are not related to source code changes and vice versa. For these reasons, we plan a more controlled experiment with the 4Conference system as part of our next software engineering course. If students find a failure, our student assistants, which are former developers of 4Conferences, are supposed to write a reproducing test so that we can verify our metric. Even though we evaluate our developer ranking metric only with quite a number of synthetic failures, we argue that its results are already satisfactory with respect to recommended developers.

Fifth, garbage collection was disabled during measurement to elide performance influences. In a realistic setting with enabled garbage collection, minimal slowdowns would be possible.

Finally, we rely on tests to obey certain rules of good style: e. g., they should be deterministic. Tests that do not follow these guidelines might hamper our conclusions. The tests that we used in our evaluation were all acceptable in this respect.

9 Related Work

We divide related work into three categories: spectrum-based fault localization, approaches to determine developer expertise, and back in time debugging.

9.1 Spectrum-based Fault Localization

Spectrum-based fault localization is an active field of research where passing and failing program runs are compared with each other to isolate suspicious behavior or state. Tarantula [11] analyzes and visualizes the overlapping behavior of test cases with respect to their results. At the system overview level, each statement is represented as a line of pixels and colorized with a suspiciousness score that refers to the probability of containing the defect. Later, Gammatella [21] presents a more scalable and generalized visualization in form of a tree map but only for classes. The Whither tool [29] collects spectra of several program executions, determines with the nearest neighbor criterion the most similar correct and faulty run, and creates a list of suspicious differences. AskIgor [5] identifies state differences of passed and failed test runs and automatically isolates the infection chain with delta debugging and cause transitions. A first empirical study [10] comparing these different spectrum-based approaches concludes that the Tarantula technique is more effective and efficient than the other ones. A more comprehensive study [1], investigating the impact of metrics and test design on the diagnostic accuracy of fault localization, states that similarity metrics are largely independent of test design and that the Ochiai coefficient consistently outperforms all other approaches.

All presented approaches produce ranked source code entities that are likely to include failure causes. However, as defects are rarely localized without doubt, developers have to determine the remaining results by hand. We argue that our presented test-driven fault navigation deals with this issue. It combines multiple perspectives based on already gathered suspiciousness information and supports developers in further approximating the real failure cause.

9.2 Determining Developer Expertise

Our developer ranking metric is mostly related to approaches that identify expert knowledge for development tasks. The expertise browser [20] quantifies people with desired knowledge by analyzing information from change management systems. XFinder [12] is an Eclipse extension that recommends a ranked list of developers to assist with changing a given file. A developer-code map created from version control information presents commit contributions, recent activities, and the number of active workdays per developer and file. The Emergent Expertise Locator [19] approximates, depending on currently opened files and their histories, a ranked list of suitable team members. An empirical study [6] verifies the assumption that programmer's activity indicates some knowledge of code and presents additional factors that also indicate expertise knowledge such as authorship or performed tasks. Besides common expertise knowledge, there are other approaches that focus on assigning bug reports to the most qualified developers. A first semi-automated machine learning approach [2] works on open bug repositories and learns from already resolved reports the relationship between developers and bugs. It classifies new incoming reports and recommends a few developers that have worked on similar problems before. Develect [17] applies a similar approach but it matches the lexical similarities between the vocabulary of bug reports and the diffs of developers' source code contributions.

In contrast to our developer ranking metric, previous approaches are generally applicable but their recommendation accuracy for a specific failure is limited. Other approaches consider either the entire system so that the search space is too large or they require similar failure reports which excludes new kinds of failures. Our metric restricts the search space to anomalies only. As anomalies are likely to include failure causes, we are able to recommend in many cases a suitable contact person. Although we require at least one failing test case, we think that often its implementation can be derived from bug reports.

9.3 Back in Time Debugging

To follow the infection chain from the observable failure back to its cause, back-in time debuggers allow developers to navigate an entire program execution and answer questions about the cause of a particular state. The omniscient debugger [14] records every event, object, and state change until execution is interrupted. However, the required dynamic analysis is quite time- and memory-consuming. Unstuck [9] is the first back-in time debugger for Smalltalk but suffers from

similar performance problems. WhyLine [13] allows developers to ask a set of *"why did"* and *"why didn't"* questions such as why a line of code was not reached. However, WhyLine requires a statically-typed language and it does not scale well with long traces. Other approaches aim to circumvent these issues by focusing on performance improvements in return for a more complicated setup. The trace-oriented debugger [26] combines an efficient instrumentation for capturing exhaustive traces and a specialized distributed database. Later, a novel indexing and querying technique [25] ensures scalability to arbitrarily large execution traces and offers an interactive debugging experience. Object flow analysis [16] in conjunction with object aliases also allows for a practical back in time debugger. The approach leverages the virtual machine and its garbage collector to remove no longer reachable objects and to discard corresponding events.

Compared to such tools, our PathFinder is a lightweight and specialized back in time debugger for localizing failure causes in unit tests. Due to step-wise run-time analysis, we do not record each event beforehand but rather split dynamic analysis over multiple runs. So, we can ensure fast response times and low memory consumption since only requested data is recorded [23]. Furthermore, our classified traces allow to hop into erroneous behavior directly. Without this concept, developers require more internal knowledge to isolate the infection chain and to decide which path to follow.

10 Conclusion

We propose *test-driven fault navigation* as an interconnected guide for debugging failures reproducible via test cases. Based on the integration of spectrum-based anomalies and failure causes, we introduce a systematic breadth-first search that navigates developers to failure causes within structure, the development team, and behavior. With the help of our corresponding tool suite, including *PathMap* and *PathFinder*, developers can localize suspicious system parts, learn about other developers who are likely able to help, and debug erroneous behavior back to failure-inducing origins. Our evaluation and case study demonstrate that *test-driven fault navigation*, combining unit testing, spectrum-based fault localization, and our step-wise run-time analysis, is practical for bringing developers closer and faster to defects. Thus, we expect that our approach is able to reduce debugging costs, especially, in agile development projects with a high ratio of testing [3].

Future work is two-fold. First, our approach will be extended to take state information into account in such a way that invariants of passing tests reveal further anomalies in the infection chain. Second, we are planning a larger user study to assess how test-driven fault navigation improves debugging in general.

References

[1] Abreu, R., Zoeteweij, P., Golsteijn, R. and van Gemund, A. J.: A Practical Evaluation of Spectrum-based Fault Localization, *J. Sys. Soft.*, Vol. 82, No. 11(2009), pp. 1780–1792.

[2] Anvik, J., Hiew, L. and Murphy, G. C.: Who Should Fix this Bug?, in *Proc. ICSE*, 2006, pp. 361–370.

[3] Beck, K.: *Test-driven Development: By Example*, Addison-Wesley Professional, 2003.

[4] Brant, J., Foote, B., Johnson, R. and Roberts, D.: Wrappers to the Rescue, in *Proc. ECOOP*, 1998, pp. 396–417.

[5] Cleve, H. and Zeller, A.: Locating Causes of Program Failures, in *Proc. ICSE*, 2005, pp. 342–351.

[6] Fritz, T., Murphy, G. C. and Hill, E.: Does a Programmer's Activity Indicate Knowledge of Code?, in *Proc. ESEC-FSE*, 2007, pp. 341–350.

[7] Gschwind, T. and Oberleitner, J.: Improving Dynamic Data Analysis with Aspect-Oriented Programming, in *Proc. CSMR*, 2003, pp. 259–268.

[8] Haupt, M., Perscheid, M. and Hirschfeld, R.: Type Harvesting - A Practical Approach to Obtaining Typing Information in Dynamic Programming Languages, in *Proc. SAC*, 2011, pp. 2169–2175.

[9] Hofer, C., Denker, M. and Ducasse, S.: Design and Implementation of a Backward-in-Time Debugger, in *Proc. NODe*, 2006, pp. 17–32.

[10] Jones, J. A. and Harrold, M. J.: Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique, in *Proc. ASE*, 2005, pp. 273–282.

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

[12] Kagdi, H., Hammad, M. and Maletic, J.: Who Can Help Me with this Source Code Change?, in *Proc. ICSM*, 2008, pp. 157–166.

[13] Ko, A. J. and Myers, B. A.: Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior, in *Proc. ICSE*, 2008, pp. 301–310.

[14] Lewis, B.: Debugging Backwards in Time, in *Proc. AADEBUG*, 2003, pp. 225–235.

[15] Liblit, B., Naik, M., Zheng, A. X., Aiken, A. and Jordan, M. I.: Scalable Statistical Bug Isolation, in *Proc. PLDI*, 2005, pp. 15–26.
[16] Lienhard, A., Gîrba, T. and Nierstrasz, O.: Practical Object-Oriented Back-in-Time Debugging, in *Proc. ECOOP*, 2008, pp. 592–615.
[17] Matter, D., Kuhn, A. and Nierstrasz, O.: Assigning Bug Reports Using a Vocabulary-based Expertise Model of Developers, in *Proc. MSR*, 2009, pp. 131–140.
[18] McCabe, T.: A Complexity Measure, *IEEE Trans. Soft. Eng.*, Vol. 2(1976), pp. 308–320.
[19] Minto, S. and Murphy, G. C.: Recommending Emergent Teams, in *Proc. MSR*, 2007, pp. 5–14.
[20] Mockus, A. and Herbsleb, J. D.: Expertise Browser: A Quantitative Approach to Identifying Expertise, in *Proc. ICSE*, 2002, pp. 503–512.
[21] Orso, A., Jones, J. and Harrold, M. J.: Visualization of Program-Execution Data for Deployed Software, in *Proc. SoftVis*, 2003, pp. 67–76.
[22] Perscheid, M., Cassou, D. and Hirschfeld, R.: Test Quality Feedback: Improving Effectivity and Efficiency of Unit Testing, in *Proc. C5*, 2012, pp. 60–67.
[23] Perscheid, M., Steinert, B., Hirschfeld, R., Geller, F. and Haupt, M.: Immediacy through Interactivity: Online Analysis of Run-time Behavior, in *Proc. WCRE*, 2010, pp. 77–86.
[24] Perscheid, M., Tibbe, D., Beck, M., Berger, S., Osburg, P., Eastman, J., Haupt, M. and Hirschfeld, R.: *An Introduction to Seaside*, Software Architecture Group (Hasso-Plattner-Institut), 2008.
[25] Pothier, G. and Tanter, E.: Summarized Trace Indexing and Querying for Scalable Back-in-Time Debugging, in *Proc. ECOOP*, 2011, pp. 558–582.
[26] Pothier, G., Tanter, E. and Piquer, J.: Scalable Omniscient Debugging, in *Proc. OOPSLA*, 2007, pp. 535–552.
[27] Queinnec, C.: The Influence of Browsers on Evaluators or, Continuations to Program Web Servers, in *Proc. ICFP*, 2000, pp. 23–33.
[28] Rawlinson, G. R.: *The Significance of Letter Position in Word Recognition*, PhD Thesis, University of Nottingham, 1976.
[29] Renieres, M. and Reiss, S.: Fault Localization with Nearest Neighbor Queries, in *Proc. ASE*, 2003, pp. 30–39.
[30] Richardson, D. J. and Thompson, M. C.: An Analysis of Test Data Selection Criteria Using the RELAY Model of Fault Detection, *IEEE Trans. Soft. Eng.*, Vol. 19(1993), pp. 533–553.
[31] Shneiderman, B.: Tree Visualization with Tree-Maps: 2-D Space-Filling Approach, *ACM Trans. Graph.*, Vol. 11, No. 1(1992), pp. 92–99.
[32] Steinert, B., Perscheid, M., Beck, M., Lincke, J. and Hirschfeld, R.: Debugging into Examples: Leveraging Tests for Program Comprehension, in *Proc. TestCom*, 2009, pp. 235–240.
[33] Vessey, I.: Expertise in Debugging Computer Programs: A Process Analysis, *Int. J. Man Mach. Stud.*, Vol. 23, No. 5(1985), pp. 459–494.
[34] Zeller, A.: *Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufmann, 2006.

Michael Perscheid

Michael Perscheid (michael.perscheid@hpi.uni-potsdam.de) is a research assistant in the Software Architecture Group at the Hasso-Plattner-Institute (HPI) and a member of the HPI Research School on Service-Oriented Systems Engineering. He received a master's degree from the Hasso-Plattner-Institute, University of Potsdam, Germany.

Michael Haupt

Michael Haupt (michael.haupt@oracle.com) is a member of the Maxine team at Oracle Labs. The work described in this paper was performed while he was a postdoctoral research assistant and lecturer in the Software Architecture Group at the Hasso-Plattner-Institute (HPI). Michael holds a doctoral degree from Technische Universität Darmstadt, Germany.

Robert Hirschfeld

Robert Hirschfeld (robert.hirschfeld@hpi.uni-potsdam.de) is a Professor of Computer Science at the Hasso-Plattner-Institut (HPI) at the University of Potsdam. He received a Ph.D. in Computer Science from the Technical University of Ilmenau, Germany. See also http://www.hpi.uni-potsdam.de/swa/.

Hidehiko Masuhara

Hidehiko Masuhara is an Associate Professor at Graduate School of Arts and Science, the University of Tokyo. He received D.Sc. from Department of Information Science, the University of Tokyo.