Follow the Path: Debugging State Anomalies along Execution Histories

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

***Abstract*—To understand how observable failures come into being, back-in-time debuggers help developers by providing full access to past executions. However, such potentially large execution histories do not include any hints to failure causes. For that reason, developers are forced to ascertain unexpected state properties and wrong behavior completely on their own. Without deep program understanding, back-in-time debugging can end in countless and difficult questions about possible failure causes that consume a lot of time for following failures back to their root causes.**

In this paper, we present *state navigation* as a debugging guide that highlights unexpected state properties along execution histories. After deriving common object properties from the expected behavior of passing test cases, we generate likely invariants, compare them with the failing run, and map differences as state anomalies to the past execution. So, developers obtain a common thread through the large amount of run-time data which helps them to answer what causes the observable failure. We implement our completely automatic state navigation as part of our test-driven fault navigation and its Path tools framework. To evaluate our approach, we observe eight developers during debugging four non-trivial failures. As a result, we find out that our state navigation is able to aid developers and to decrease the required time for localizing the root cause of a failure.

***Index Terms*—Back-in-time Debugging, Likely Invariants, Dynamic Analysis, Testing, Test-driven Fault Navigation**

I. Introduction

Debugging is largely an attempt to understand what causes failures [1]. Starting with a test case, which reproduces the observable failure, developers follow failure causes and their effects on the infection chain back to the root cause (defect). For localizing failure causes, they examine involved program entities and distinguish relevant from irrelevant behavior and expected from unexpected state. After understanding all details of failure causes and their effects, developers are able to identify and correct the root cause. Unfortunately, this idealized procedure requires deep knowledge of the system and its behavior [2]. Since failures and defects can be far apart from each other, their unknown infection chains are consequently long and demand a laborious effort for debugging [3].

To better comprehend what causes failures, back-in-time debuggers [4] can help developers by providing access to all required execution details. These debugging tools include every information for describing what happened before observable failures. However, they also force developers to understand a large amount of run-time data that bears little relation to failure causes. As there are no hints to unexpected state properties or misleading behavior, developers have to decide what is right or wrong. Especially, the missing identification of suspicious state makes tracing of infection chains a tedious task. At each suspect position in the execution history, developers have to examine the current object space for potential failure causes and find the related computation completely on their own. We state our research question as follows:

> How can we support developers in localizing corrupted program states, tracing of infection chains, and understanding how failures come into being?

We argue, if developers get hints about suspicious program entities and how they are related to each other, they are able to systematically trace failures back to their root causes. They understand failure causes step by step, make fewer and simpler decisions about how to follow execution histories, and focus their attention on finding defects [1].

In this paper, we present *state navigation* as a debugging guide that reveals infection chains by emphasizing state anomalies along execution histories. Starting with deriving common object properties from passing test cases, we are able to generalize expected program states. Based on such likely invariants, we generate dynamic contracts and compare them with failing test cases. Differences reveal state anomalies that can be good indications of failure causes. With a mapping on execution histories of our previous back-in-time debugger [5], [6], anomalies reveal infection chains and guide developers to root causes. Thus, our state navigation supports developers in understanding failures and how they come into being.

The contributions of this paper are as follows:

- *State navigation* integrates likely invariants into a back-in-time debugger to reveal unexpected state properties and to assist in the observation of infection chains.
- *Inductive analysis* derives likely invariants by incrementally generalizing common object properties such as types and value ranges from methods that are covered by passing tests and overlap with the failing execution.
- *The Path tools framework* implements our completely automatic approach as part of our test-driven fault navigation in Squeak/Smalltalk [7].

To evaluate our state navigation, we present a motivating example and conduct a comparative user study within the

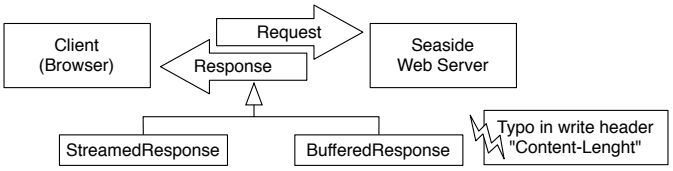

Fig. 1. An inconspicuous typo corrupts the header state of buffered responses and leads to faulty results of several client requests.

scope of a real world project. In this case, we discover that our approach is able to further decrease debugging cost with respect to required time and developer's effort.

The remainder of this paper is structured as follows: Section II introduces the background of our approach. Section III presents our state navigation. Sections IV explains our inductive analysis. Section V outlines our Path tools framework. Section VI evaluates our approach, Section VII discusses related work and Section VIII concludes.

II. Finding Causes of Reproducible Failures

We introduce a motivating example that serves as a basis for our discussion of debugging challenges, our test-driven fault navigation[1], and the explanation of our approach in the following sections.

A. Motivating Example: Typing Error in Seaside

We have inserted a defect into the Seaside Web framework [8] and its request/response processing logic (`BufferedResponse` class, `writeHeadersOn:` method). Fig. 1 illustrates the typing error inside the header creation of buffered responses. The typo in "Content-Lenght" is inconspicuous but corrupts the header state. For that reason, browser requests that demand buffering are unable to render the invalid 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 [9].

B. Challenges of Debugging

However, debugging this example and other failures faces several challenges with respect to localizing root causes.

1) Symbolic Debugger: How to Follow Infection Chains?: Although symbolic debuggers are nowadays available in almost all development environments, they are not well-suited for systematically following infection chains back to their root causes. They only allow developers to stop a program, step forward, and to access the run-time stack at a particular point in time. They neither report what is going wrong nor offer capabilities to go back in time. In our motivating example, developers only have access to the last point in execution without any hints to the corrupted state in buffered responses and no possibility to access the prior execution. Thus, developers need to rely primarily on their intuition in order to understand and find past failure causes with a tool that is only be able to debug in the forward direction.

2) Back-in-time Debuggers: What Happened before?: In contrast to symbolic debuggers, back-in-time debuggers [4] provide access to entire execution histories by recording all run-time information before the failure occurs. Based on this data, developers can start with the observable failure, step backward, and search for failure causes at each point in the program's execution history.

However, starting debugging at failures still includes a long way back to their root causes [3]. Back-in-time debuggers do not emphasize failure causes and so developers have to examine an enormous amount of data on their own. Especially, the missing classification of suspicious and harmless program entities leads to numerous and often laborious decisions which execution subtree to follow [1]. In our motivating example, a back-in-time debugger records every method call and each state change during a request-response processing. Although this information contains everything that is required for localizing the root cause, developers have no support for tracing the infection chain within the large amount of run-time data. Thus, debugging entire executions can become a tedious activity.

3) Anomalies: What Are Possible Failure Causes?: Anomalies such as from program spectra [10] or likely invariants [11] automatically identify possible failure causes by deriving run-time properties from reference runs and comparing them with failing executions. Differences have a high probability of including failure causes [12] and so help developers to narrow down the search space by answering which program entities are potentially infected.

Unfortunately, existing approaches are not well-suited for localizing root causes because they potentially detect false positives, distribute loosely-coupled anomalies all over the program, and do not relate suspicious properties with failing behavior [13]. For each difference, developers still have to answer whether the anomaly is a failure cause and, if so, how it is related to the infection chain. For these reasons, developers have only a lot of unrelated starting points that must tediously be debugged one by one. In our motivating examples, there are several anomalies all over the Seaside Web framework. Although one of them also includes a typo close by the defect, developers need a lot of time to costly debug unrelated anomalies one after the other.

C. Test-driven Fault Navigation

Our test-driven fault navigation [5] is a debugging guide that integrates spectrum-based anomaly detection [10], [12] into a systematic breadth-first search for tracing failure causes back to defects. Starting with at least one test case that reproduce the observable failure, we localize anomalies by comparing the method coverage of all failed and passed test cases. Methods being executed by a large number of failing but only a few

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

passing tests have a higher failure cause probability (anomaly) than methods being executed by less failing but many passing test cases. By integrating these spectrum-based anomalies into our back-in-time debugger [6], we highlight suspicious method calls and allow developers to distinguish between suspicious and expected run-time behavior. Thus, our execution histories include additional information about failure cause probabilities that gives developers helpful advice on how to follow infection chains back to their root causes.

Fig. 2 (a) illustrates the integration of spectrum-based anomalies into execution histories of failing test cases. The small example represents the infection chain with the observable failure (method 11, bottom right corner) and the root cause also known as defect (method 4, center left). Each row shows all eleven methods and highlights the specific method that is executed at this point in time. Colors express the failure cause probability (from red as high to green as low) which are computed by the comparison with the passing test case. In the end, the execution history is classified with anomalies and developers can directly start debugging on the left sub tree—as methods 8 and 6 are less suspicious, developers can shorten the infection chain to methods 5, 4, and 2. Without this anomalous guidance, it is hard to abbreviate the trace because developers have to decide at each executed method whether there is a failure cause or not.

Although our test-driven fault navigation already leads developers along suspicious *behavior*, it misses important information about corrupted *state*. At this point, developers cannot easily answer the critical question "which program state is infected?" because the applied spectrum-based anomalies are only based on behavioral coverage data. In our motivating example, we guide developers close to the defect but the analysis of the unexpected header state and the difficult identification of the typo remains open. As our previous approach misses to highlight corrupted state properties along infection chains, it lasts unclear what causes the failure.

III. State Navigation

Our state navigation technique reveals infection chains by identifying corrupted state along execution histories. To detect state anomalies, we first derive likely invariants from passing test cases and then create dynamic contracts that verify the collected object properties. Differences during the execution of failing test cases uncover state anomalies that have a high probability to include failure causes. To reveal infection chains and explain how corrupted state is related to each other, we emphasize anomalies along the failing behavior. Our entire state navigation works automatically and requires only a comprehensive test suite with several passing test cases.

A. Likely Invariants from Passing Test Cases

We start our state navigation by deriving likely invariants. To learn common object properties, we first execute all passing test cases and collect their used objects. For each method called, we check its arguments, the return value, and the receiver object. Having a concrete object, we accumulate its specific properties into generalized invariants that hold with previously observed objects at the same program entity. We analyze several properties such as type information and value ranges. During this analysis, we steadily expand likely invariants once concrete objects come along with new properties that are not covered so far. Finally, we gather enough data for pre- and post-conditions of methods as well as invariants of accessed instance variables.

In Fig. 2 (b), we illustrate the concrete objects of a passing test case example. In different executions, method 6 is called with 1.0, 5.2, and 10 at the same argument. During its first method call, we collect a `Float` in the value range of 1.0 to 1.0. With the second execution, we extend the value range to 5.2. Finally, with the last `Integer` argument we change the type to the superclass `Number` and enlarge the value range again. Thus, we have a generalized object property for this argument that expects numbers between 1 and 10. We also derive invariants by all other called method. The arguments of method 5 are equal to method 6, while its return values allow `Integer` types between 1 and 30. These return values are input to method 8 that always returns a receiver object with the same specific class type. All observable objects of method 9 are the same booleans and for that reason they also uncover a possibility for writing more tests.

Although the collected information is helpful with respect to program comprehension and later debugging, its quality strongly depends on the test base. We assume that concrete objects have meaningful values and that tests check the entire code base comprehensively. The first assumption is valid because test cases should check at least common use cases of a system. So, their execution includes the most important state properties for program comprehension and our likely invariants. The second assumption depends on the test coverage of the system. If test cases do not check extensive enough, we miss invariants or they tend to be too specific. To provide meaningful results, we assume systems that test again the most common use cases and so reach a test coverage above 50 %. In the case of too specific invariants, we argue that this concrete data is at least a valuable source for understanding the expected program behavior.

B. Detection of State Anomalies with Dynamic Contracts

With likely invariants, our state navigation generates dynamic contracts [14] that automatically detect corrupted state in failing test cases. For each generalized object property, we create suitable assertions and aggregate them into contracts. Depending on the covered program entity, we add contracts for arguments as pre-conditions to methods, return values as post-conditions to methods, and receiver state as invariants to their corresponding classes. After that, developers are able to run test cases with activated contracts. While passing test cases should work correctly, failing test cases create anomalies if their state is not in accordance to our expected invariants. Such state differences have a high probability to include failure causes so that developers can rely on this information for creating their hypotheses [1]. To further reveal parts of the

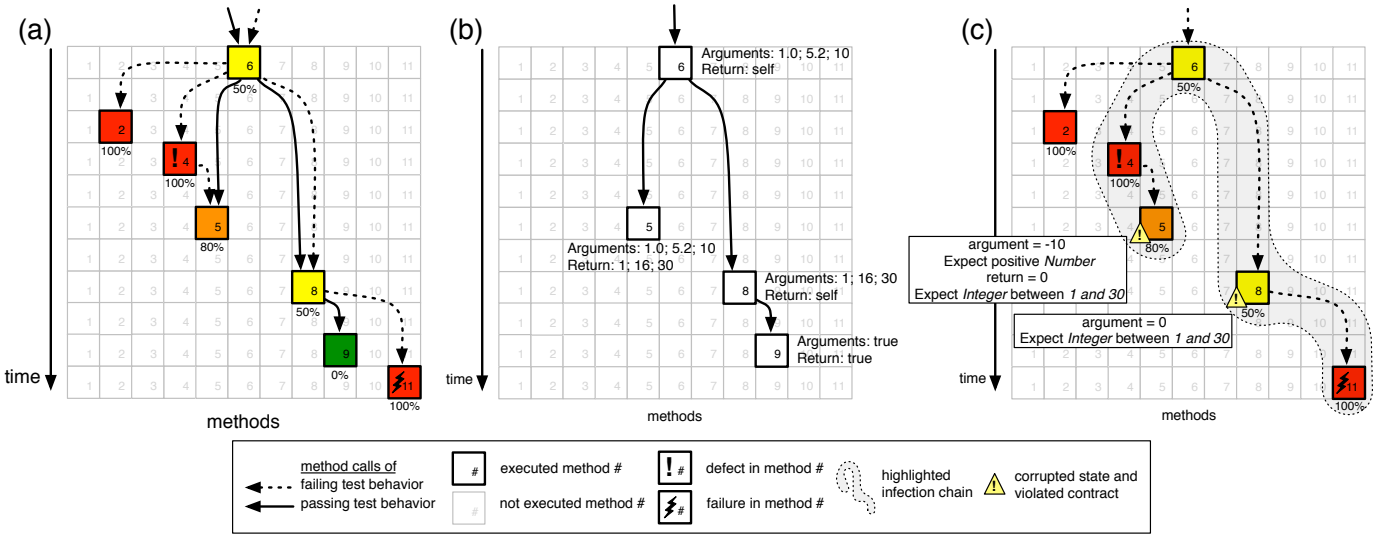

Fig. 2. Test-driven fault navigation with its suspicious execution history (a) is extended with our state navigation. Based on the derivation of likely invariants from passing test cases (b), we reveal state anomalies and emphasize parts of the infection chain within the failing behavior (c).

infection chain and to understand how anomalies relate to each other, we map these state violations to the execution history as provided by a back-in-time debugger. Thus, developers can explore the failing behavior and our state navigation guides them along the infection chain.

In Fig. 2 (c), we present how state anomalies complete our test-driven fault navigation by revealing parts of the infection chain. Based on the derived invariants, our failing test case calls method 5 with -10 as argument and violates the contract for positive numbers. After that the method returns an unexpected 0 and violates its postcondition as well as the precondition of method 8. While the `Integer` type is still valid, the value range assertion allows only numbers between 1 and 30. All state anomalies are valuable indications for failure causes and with their mapping along the suspicious execution history, it further reveals parts of the infection chain. Developers are able to understand corrupted state properties and how to follow the observable failure back to its root cause.

C. Example: Coming Closer to the Typing Error

In our Seaside typing error, we first run all passing tests from the still working streamed responses and collect type and value ranges of their applied objects. For example, we verify string arguments with a spellchecker. Based on this data, we derive likely invariants, create corresponding contracts, and so propagate the implicit assertions of the response tests to each covered method. Finally, we execute our failing test case with enabled contracts. As soon as a contract is violated, we mark its exception in the execution history and proceed. Among others, we reveal two state anomalies close by the defect.

Fig. 3 summarizes the call tree of our back-in-time debugger [5], [6] with our state navigation extension. We mark method calls that trigger a violation with small exclamation marks (1). Developers can further inspect these violations and see that a precondition fails in

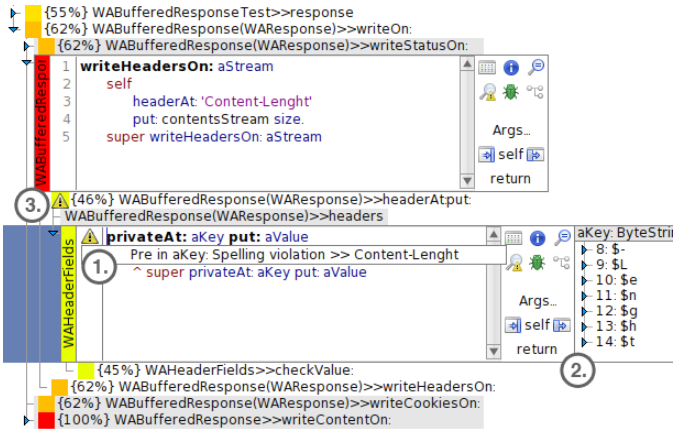

Fig. 3. State anomalies (exclamation marks) extend our back-in-time debugger to highlight the typing error and to reveal the infection chain near the defect. (Colors relate to our existing test-driven fault navigation and reflect the failure cause probability of method calls from red (high) to green (low))

`WAHeaderFields>>privateAt:put:`. There is a spelling violation in the first argument of this method because all streamed responses and their passing test cases used correctly spelled identifier keys for their header information. The corrupted state is opened for further exploration on the right (2). As our typo in "content-lenght" is automatically revealed, our state navigation gives developers helpful advice about the real failure cause. Another spelling violation is close by and with the help of our previous test-driven fault navigation developers can easily follow the infection chain back to the root cause in the very suspicious and red-colored `writeHeadersOn:` method (3).

IV. Inductive Analysis

For our state navigation, we need to derive common object properties from passing test cases. However, the correspond-

ing dynamic analysis of likely invariants tends to be time-consuming because each object has to be explored at each execution point in full detail [11]. Especially, in large systems this method is not only expensive but also results in a vast amount of data with numerous imprecise invariants. Both drawbacks limit the debugging possibilities of current approaches [13]. For that reason, our inductive analysis restricts uncertain assertions and collects object properties only if needed. First, we analyze only common object properties with unambiguous assertions such as types and value ranges to limit false positives and provide reliable state anomalies. Second, we derive as few as possible likely invariants for the current failure. We limit the initial analysis scope to methods that are only executed in failing and passing test cases. Depending on their needs, developers can control the kind of collected object properties. If desired, they can complement the results later on by re-executing test cases again and at the same time focusing on other object properties and methods. Fig. 4 summarizes our inductive analysis approach.

On the left, developers first select the system packages for our analysis. Based on the coverage of our failing tests, we instrument each of the included classes and their methods with *harvester wrappers* to transparently add analysis code around the original implementation [15]. After that, we execute all passing test cases that overlap in method coverage with failing test cases. If a wrapped method is executed, we send object events including concrete values to our inductive analysis tracer. In the example of Fig. 4, developers are only interested in the suspicious `BufferedResponse` class. If a passing test case later calls the wrapped method `initializeOn:`, we send the concrete `ReadWriteStream` object as its first argument to our inductive analysis tracer.

In the middle, our *inductive analysis tracer* receives object events and forwards them to its *harvesters* for generalizing object properties. Depending on the developer's choice, our tracer includes several harvesters for different properties. Each harvester notes common object properties in so called *buckets* to compare and align upcoming object events with them. Each bucket consists of dictionaries for generalized method arguments, return values, and instance variables and a mapping to their program entities. In Fig. 4, developers indicate interest in type and value range properties. While the first harvester collects from objects their most common superclass type, the latter explores value ranges of primitive objects such as numbers, strings, and streams. In the type bucket, we find for the first argument of `initializeOn:` the common `Stream` superclass. In the range bucket, we store stream properties such as data about its content and if it is already closed.

On the right, our derived object properties can be accessed by our state navigation to create dynamic contracts. We iterate over all generalized object properties and create assertions as source code snippets. With a contract builder, we aggregate all assertions into corresponding contracts for pre- and postconditions and invariants. For example, Fig. 4 presents the created source code for the `Stream` type and its value range which is then added to the `initializeOn:` method as contract.

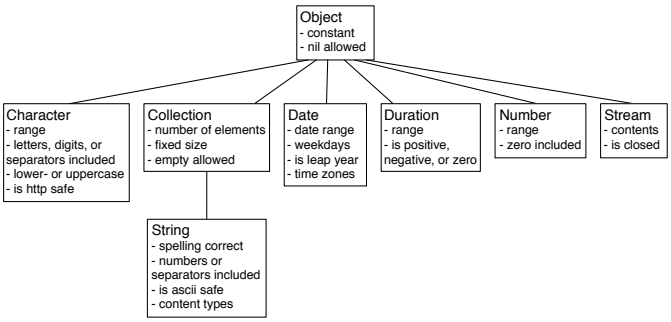

Fig. 5. Harvested value ranges of primitive Smalltalk objects.

Based on this architecture, we can implement arbitrary harvesters for collecting generalized object properties. Developers only have to specify how common properties are derived from concrete objects and stored into the bucket. If this new harvester is installed in our inductive tracer, object events automatically call corresponding interface methods for arguments, return values, and instance variables. Furthermore, each object property should implement a `printContract` method that automatically converts its generalized data into assertions. So far, we have implemented two specific harvesters.

A. Type Harvester: Collecting Type Information

Our type harvester gathers detailed type information of executed program entities [16][2]. This is especially helpful in dynamically typed programming languages such as Smalltalk where type information is not explicitly represented in source code. From each covered method argument, return value, and instance variable our type harvester derives its most common superclass by checking the type of the concrete object and comparing it to the deposited information in our type bucket. In the case that our bucket does not know the related program entity, we note the new type. If both types are equal or the new type inherits from the stored type, we do nothing because the stored information already comprises the new type. In all other cases, we store the common superclass of both types. Furthermore, we also collect type information of container objects and their children such as in collections and dictionaries. For creating dynamic contracts, we provide a generic implementation of the `printContract` method at Smalltalk's root class. With some meta-programming, this method creates the corresponding type assertion for all classes.

B. Range Harvester: Checking Value Ranges of Objects

Value range harvesting collects common object properties for primitive data types such as numbers, collections, and strings. Fig. 5 summarizes all properties for primitive objects that we currently harvest. For example, in the case of a number we store its value range and check if it contains a zero. If an object's class inherits from other primitive types, we check

[2]Compared to our previous work, we improve our type harvester with respect to generalizability (inductive analysis tracer), performance (method wrappers), support of containing types, and creation of contracts.

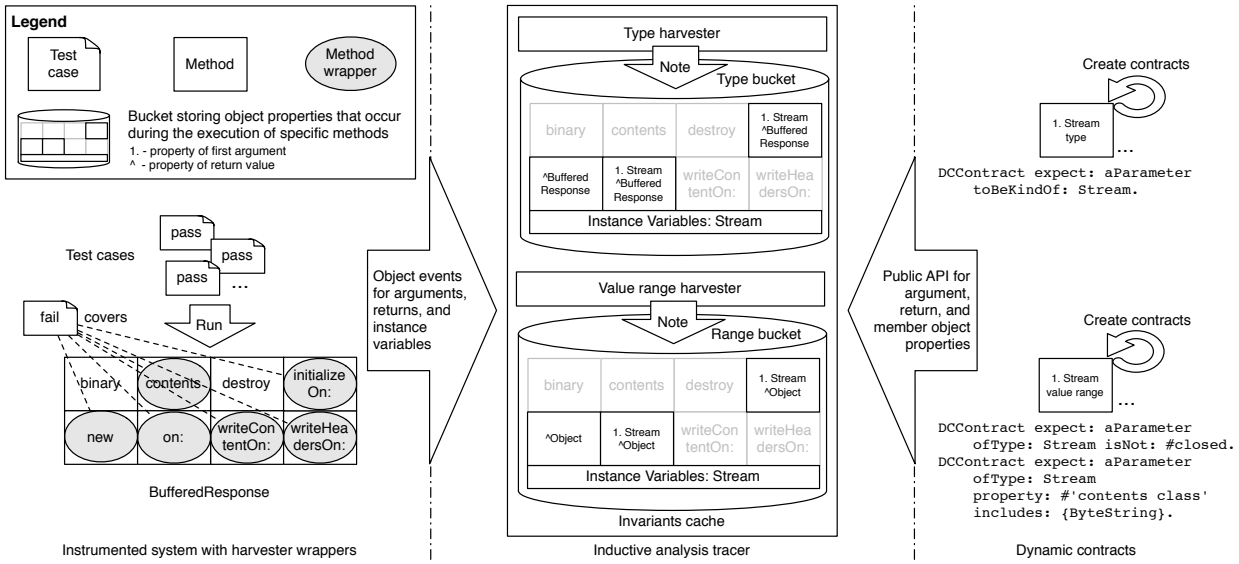

Fig. 4. We derive common object properties from passing test cases with method wrappers and buckets in order to create dynamic contracts later on.

all of the corresponding properties, too. Since numbers inherit from objects, we also harvest whether numbers are constant and contain `nil` values.

To derive all these common properties, we first check which behavior the corresponding object includes and then collect the common object properties with Smalltalk's libraries and reflection mechanisms. For example, the *spelling correct* property is verified by a spellchecker. If common object properties do not exist, we create a new range property for this concrete object. Otherwise, we compare the new and existing value ranges and, if necessary, expand the common properties. For example, we assume that `nil` objects are not allowed, but as soon as an object event includes an undefined object we change this value to `true`. In this case, the object property denotes a final value that cannot be changed by subsequent events without `nil` objects. In this way, we generalize object properties step by step. Finally, each range property implements the `printContract` method and knows how to convert its generalized data into proper assertions.

C. Discussion

So far, our inductive analysis only derives types and value ranges. We started with this small subset of likely invariants because we do not want developers to be overwhelmed by countless violated contracts. Otherwise, developers would need more time for comprehending assertions instead of following infection chains backwards. We argue that already such a small subset can help in debugging as it restricts the number of violations, false positives, and unimportant invariants. Compared to Daikon [11] that provides a considerable amount of more likely invariants, it has been shown that too many invariants can limit their intent for debugging [13], [17]. Nevertheless, we consider the incremental integration and evaluation of Daikon's rich set of likely invariants for our debugging purposes as future works.

V. The Path Tools Framework

We implement our state navigation as part of our test-driven fault navigation and its corresponding Path tools framework [5] for the Squeak/Smalltalk development environment [7]. The tool suite consists of our enhanced test runner PathMap [18] and our lightweight back-in-time debugger PathFinder [6]. The Path analysis framework provides the basis for our tools on top of the SUnit testing framework[3]. By leveraging unit tests as a basis for dynamic analysis, we can ensure reproducibility and a high degree of automation, scalability, and performance during debugging with our tools.

To support our state navigation, we extend our PathMap with an additional flap for deriving object properties and adding contracts. With that developers only have to select test suites and harvesters—everything else is done automatically. Furthermore, we enhance our PathFinder to react on contract violations and present the corresponding state anomalies along execution histories. While colors represent spectrum-based anomalies, exclamation marks emphasize state anomalies (compare to Fig. 3). For a detailed description of our tools and how to debug with them, we refer to [19].

VI. Evaluation

We conduct a user study and compare developers while debugging with standard and our Path tools. As a result, we find out that our state navigation can further decrease the required time for localizing the root cause of a failure.

A. Experimental Setup

We base this study on our previous user study [5] in order to assess the positive influence of our state navigation. It is difficult to evaluate the presented approach in isolation because

[3]We consider unit test frameworks as a technique for implementing different kinds of test cases such as acceptance, integration, and module tests.

TABLE I
DESCRIPTION OF ICALENDAR'S FAILURES.

Failure	Description	Difficulty	Infection length	State anomalies
1	Unintended string constant in phone types	Normal	5	1
2	Forgotten deletion of obsolete calendar events	Normal	84	18
3	Missing separator for parsing event files	Hard	520	28
4	Return of improper but polymorph objects for alarms	Hard	2133	1

of its integration into our complete test-driven fault navigation and its corresponding Path tools. For that reason, we examine whether other developers with a similar background can debug the same hard to solve failures in less time.

We choose Squeak's iCalendar project[4] 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 file import and export functionality 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, comprehensive test base, understandable domain, and ideal project size (77 classes, 1,375 methods, 7,704 lines of code, and 71.42 % test coverage) that is neither too small nor too large.

For our user study, we observe eight developers that have a similar background as the participants of our previous study [5]. All of them are computer science students in the 6th-8th semester, with about six years of programming experience and professional expertise with symbolic debuggers. They are well acquainted with object-oriented programming and Smalltalk because they passed our software engineering courses with excellent grades. All participants have similar development skills and become familiar with iCalendar in this 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, our participants are supposed to localize four failures which we have already applied in our previous study[5] [5]. Table I describes their characteristics. We insert these four defects all over the system, whereby we described them obviously. For example, we comment important statements instead of deleting them. As we focus on following infection chains, this help simplifies the final verification a bit but not the more time-consuming localization. For each failure, we assess a difficulty level that estimates the required debugging effort based on our experience and the infection chain length. All failures are reproducible in 1-10 failing tests, which do not trivially include defects within their stack traces.

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

[5]Compared to our previous study, we do not observe the two easy failures because our approach already localized them in a short amount of time.

B. User Study Procedure

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

We conducted the user study by observing our developers while debugging iCalendar's failures with different tools. First, all developers debugged two failures with Squeak's standard debugging tools. After that, they debugged the remaining two failures with our Path tools. While using test-driven fault navigation, we provided assistance with handling PathFinder's user interface and its features. The process of localizing failures remained free of influence. For all four failures, we measured the complete debugging time. If the defect was not localized after 15 minutes, we marked the failure as not solved.

To evaluate our approach, we assigned each developer two failures for debugging with standard tools (symbolic debugger and test runner) and two failures for test-driven fault navigation (PathFinder and PathMap). For each of the four failures, we ensured a unique combination of developers and applied tools. At each level of difficulty, a developer debugged one failure with standard and the other one with our Path tools.

After the study, we interviewed each participant and asked for feedback with respect to our approach and Path tools.

C. Discussion of Study Results

Fig. 6 summarizes the required debugging time for each failure with standard tools and our Path tools. For both levels of difficulty, the position of a bar corresponds to the same developer with respect to the applied tools per failure. For example, the second bar in normal failure 1 standard debugging and the second bar in normal failure 2 test-driven fault navigation represent the same developer.

In the case of failure 1, developers with test-driven fault navigation are very fast compared to developers with symbolic debuggers and test runners. While our approach requires less than one minute each time, standard debugging tools range from about two minutes to not solved after 15 minutes. Regarding our state navigation, we reduce the debugging time by additional two minutes because the state anomaly matches the defect very close. One method call after the defect, a violation indicates the unexpected string as an obvious cause.

With failure 2, developers with standard tools have some problems in localizing failure causes while our state navigation performs very well. Two standard debugging students do not find the defect within 15 minutes and also the other two require at least the same time as the slowest test-driven fault navigation participant. The remaining three students just need about three minutes and test-driven fault navigation is once more faster. However, state navigation adds one more minute than in our previous study. We observe two reasons for this issue. First, even if we expected similar development skills, we

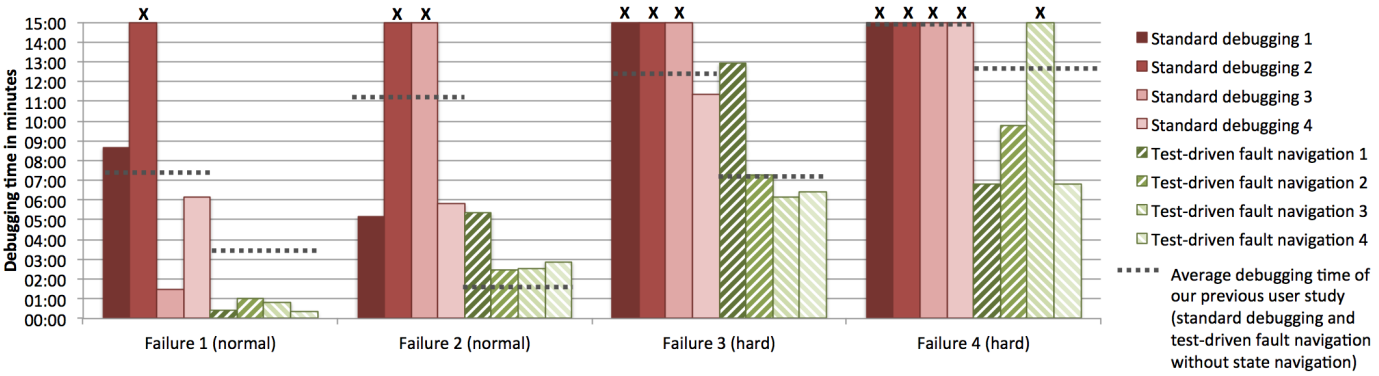

Fig. 6. Required debugging time (the less the better, x marks not solved failures) with Smalltalk's standard tools (symbolic debugger and test runner) compared to our test-driven fault navigation including state navigation and its corresponding Path tools (PathFinder and PathMap).

see a few differences between the participants for the benefit of the previous study—the standard debugging time is in many cases above the previous average. Second, developers need more time for understanding state anomalies in comparison to the suspicious method calls of our behavior navigation. For example, developers analyze a size violation for a while before they comprehend which items are wrong. But this analysis also allows them later on to explain what causes the failure.

The hard failure 3 is solved by only one developer with standard tools and all with our Path tools. Besides the slowest test-driven fault navigation participant with 13 minutes, all other identify the defect within six to seven minutes and solve it faster than in our previous study. Although only two students are one minute faster, we argue that our state navigation still helps a lot. Compared to our previous study, all participants explain the root cause and the infection chain very well. Thus, they could fix the defect without any problems.

Failure 4 is so difficult that no developer solves it with standard tools but three out of four with our approach. With our Path tools, two developers find this failure in less than seven minutes and another one in about ten minutes. With our state navigation, all three developers are several minutes faster than in our previous study. However, one student does not solve this failure with our tools. Although he followed the infection chain close to the defect, he did not understand how the state anomaly relates to the root cause. Due to the wrong polymorphic receiver object, the state anomaly suggested a correct argument (for another receiver) as corrupted. As the student focused his debugging effort on the complex argument, he could not solve this failure in time.

During debugging, we observed the participants and noticed some interesting insights. Developers with standard tools relied primarily on their intuition. Often, they guessed reasons for failure causes such as wrong behavior and infected state. In doing so, some developers had proper hypotheses about failure causes, but no participant was consistently better at guessing than another one. This observation was also reflected in the differences of required debugging time. In contrast, our test-driven fault navigation allowed developers to rely on a systematic debugging process and the advice of our tools. With the help of our additional state navigation, developers linked the corresponding anomalies with their hypotheses and followed the infection chain backwards. Compared to our previous studies, they focused first on state anomalies and followed later on the suspiciousness scores of method calls. This improved in many cases not only the required debugging time but also led to reduced effort because developers got helpful advice about what went wrong.

With the help of our user study, we conclude that our state navigation can reduce the required time for localizing failure causes. Compared to standard debugging tools, developers who apply our test-driven fault navigation need in the majority of cases less time for debugging. Especially, our presented state navigation complements our debugging approach and is able to further decreases the required debugging time by several minutes. With the additional integration of state anomalies into our back-in-time debugger, developer better understood how failures come into being and could directly jump to unexpected state properties. All developers confirmed that our approach had been promising for debugging with less effort. They liked our Path tools and conceived them as very valuable for debugging. A participant summarized it as follows: *"The state anomalies are very helpful and a good advice to failure causes. In particular, the combination with a back-in-time debugger, suspicious method calls, and unexpected state properties allows me to easily follow the infection chain back to its root cause."*

D. Threats to Validity

We rely on tests to obey certain rules of good style; they should be reproducible and deterministic. Tests that do not follow these guidelines might hamper our conclusions. The tests in our evaluation were all acceptable in this respect.

The chronological order of debugging the first three failures with standard tools could positively influence participants' program comprehension. To reduce this factor, we have a preparation phase of two hours to become acquainted with iCalendar and our tools. 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.

Regarding the quality of state anomalies in the emphasis of infection chains, we expect no false positives. We consider each anomaly as valid with respect to the root cause because of our experimental setup. We derive common object properties only from overlapping passing test cases. Without a defect and activated dynamic contracts, all test cases pass and no state anomaly occurs. Thus, each contract violation is caused by this defect.

A current study reports that automatically generated program invariants by Daikon [11] are sometimes hard to understand and tend to produce false positives [17]. In comparison to the numerous, complex, and linked invariants by this tool, we limit our inductive analysis to unambiguous assertions such as types and value ranges. Our invariants are not related to each other with the effect that they are easier to understand and mostly prevent misleading results. Nevertheless, these assertions are still strong enough to emphasize several failure causes along infection chains. An integration of Daikon's more complex invariants into our debugging approach and a full discussion about benefits and drawbacks remains future work.

Even if our evaluation does not cover all kinds of defects, we applied several realistic failures that lead to numerous state anomalies. We base our applied defects on hard to solve failures that we experienced in other projects multiple times. For example, typos are common and often difficult to localize [20]. Also the remaining two easy failures of our previous user study include a few state anomalies that cover their infection chains. For that reason, we introduced these failures during our user study preparation.

As we compare our evaluation with our previous study, individual developer skills could influence our results. Due to the fact that we had no access to the previous participants anymore, we conduct a study with similar experienced developers and the same failures to analyze the benefits of our state navigation. Although we choose students with similar grades, experience, and project results, we observed by some participants weaker debugging skills than expected. Nevertheless, we can conclude that our state navigation improves debugging because the shown improvements with weaker participants probably implicate still better results with equal skills.

VII. Related Work

As our approach combines likely invariants with back-in-time debugging, related work deals with these two topics.

A. Back-in-time Debugging

To follow infection chains from observable failures back to their root causes, 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 (ODB)* [4] records every event, object, and state change until execution is interrupted. However, the required dynamic analysis is quite time- and memory-consuming. *Unstuck* [21] is the first back-in-time debugger for Smalltalk but suffers from similar performance problems. *WhyLine* [22] 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 does not scale 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 (TOD)* [23] combines an efficient instrumentation for capturing exhaustive traces and a specialized distributed database. Later, a novel indexing and querying technique [24] ensures scalability to arbitrarily large execution traces and offers an interactive debugging experience. *Object flow analysis* [25] provides a practical back-in-time debugger that leverages the virtual machine and its garbage collector to remove no longer reachable objects and to discard corresponding events.

All presented back-in-time debugger focus on presenting execution histories as efficient as possible. However, they do not classify the large amount of run-time data in order to localize failure causes more easily. For that reason, developers still have to understand long-running infection chains by themselves. Compared to these tools, our PathFinder is a lightweight and specialized back-in-time debugger for localizing failure causes in failing test cases [6]. So far, this tool already supports the classification of suspicious method calls with spectrum-based anomalies [5]. Our state navigation completes this approach by highlighting corrupted state and automatically revealing infection chains. Without our navigation concepts, developers require more internal knowledge to isolate failure causes and to decide which path to follow.

B. Likely Invariants

Likely invariants automatically identify possible failure causes by deriving run-time properties from reference runs and comparing them with failing executions. Differences have a high probability of including failure causes and so help developers to narrow down the search space by answering which program entities are potentially infected.

Daikon [11] comprises a set of automatic techniques for inferring generalized program state from execution traces. It observes occurring objects and summarizes their specific properties into invariants such as non-zero properties and containment relationships. These invariants detect not only corrupted state, but also help in generating test cases and repairing inconsistent data structures. However, the first version suffers from a scalability issue which creates too many results, misses important invariants, and requires too much time. Later, an *extension to Daikon* [26] solves this problem by optimizing polymorphism and filtering unchanged values. *Carrot* [13] experiments with a subset of Daikon's invariants and tries to localize failure causes. However, the experimental results are unsatisfactory because anomalies are distributed all over the program. Developers cannot relate violated invariants with failing behavior so that they have to tediously debug all these unrelated starting points one by one. *Diduce* [27] extends Daikon's approach and derives invariants on the fly.

During program execution, it monitors its run-time and gradually relaxes invariants of observed objects. After a while, developers receive violations only for corner cases, which then identify failure causes more easily. *Screeners* [28] further optimize the run-time overhead and decrease it to only 14 %. *ClearView* [29] also monitors invariants on the fly but it applies them to automatically patch upcoming failures. As soon as a violation occurs, it generates candidates to hold the invariant and continues the application with changed state. Finally, comparative studies [17], [30] of programmer-written and likely invariants conclude that a combination of both methods is most promising.

Our state navigation discovers likely invariants and reveals infection chains by emphasizing state anomalies along execution histories. We analyze run-time data from test cases similar to Daikon with the difference that we restrict invariants to unambiguous assertions and developers can adapt the analysis to their needs. To ensure that developers can rely on revealed state anomalies, we derive only invariants that include a low probability of false positive results. Furthermore, developers can incrementally refine the derived invariants in additional test runs and add handwritten assertions if needed. Finally, we map violated contracts on execution histories to reveal infection chains. This integration further supports developers in understanding and localizing failure causes.

VIII. Conclusion

We propose *state navigation* as new part of our test-driven fault navigation that guides developers during debugging from observable failures back to their root causes. Starting with our *inductive analysis* that derives likely invariants from passing test cases, we create dynamic contracts and compare them with failing executions. Based on violated contracts, we emphasize state anomalies along execution histories and so automatically reveal infection chains. We implement our state navigation within the *Path tools framework*. It consists of PathMap, an extended test runner for deriving likely invariants, and PathFinder, a lightweight back-in-time debugger for integrating uncovered anomalies into its execution history. With the help of a user study, we show that our presented state navigation supports developers not only in revealing infection chains but also further decreases the required debugging costs.

Future work deals with three topics. First, our approach will be extended with why-questions [22] to even better guide developers through execution histories. Second, our anomalies shall be refined with interactive developer feedback [31] in such a way that identified false positives strengthen the remaining failure cause probabilities. Finally, we are planning a larger user study to evaluate the benefits of several debugging techniques in practice.

References

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

[2] I. Vessey, "Expertise in Debugging Computer Programs: A Process Analysis," *Int. J. Man Mach. Stud.*, vol. 23, no. 5, pp. 459–494, 1985.

[3] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan, "Scalable Statistical Bug Isolation," in *PLDI*, 2005, pp. 15–26.

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

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

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

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

[8] M. Perscheid, D. Tibbe, M. Beck, S. Berger, P. Osburg, J. Eastman, M. Haupt, and R. Hirschfeld, *An Introduction to Seaside*. Software Architecture Group (Hasso-Plattner-Institut), 2008.

[9] G. Rawlinson, "The Significance of Letter Position in Word Recognition," Ph.D. dissertation, University of Nottingham, 1976.

[10] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. van Gemund, "A Practical Evaluation of Spectrum-based Fault Localization," *JOSS*, vol. 82, no. 11, pp. 1780–1792, 2009.

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

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

[13] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. Reiss, "Automated Fault Localization Using Potential Invariants," in *AADEBUG*, 2003, pp. 273–276.

[14] R. Hirschfeld, M. Perscheid, C. Schubert, and M. Appeltauer, "Dynamic Contract Layers," in *SAC*, 2010, pp. 2169–2175.

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

[16] M. Haupt, M. Perscheid, and R. Hirschfeld, "Type Harvesting A Practical Approach to Obtaining Typing Information in Dynamic Programming Languages," in *SAC*, 2011, pp. 1282–1289.

[17] M. Staats, S. Hong, M. Kim, and G. Rothermel, "Understanding User Understanding: Determining Correctness of Generated Program Invariants," in *ISSTA*, 2012, pp. 188–198.

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

[19] M. Perscheid and R. Hirschfeld, "Follow the Path: Debugging Tools for Test-driven Fault Navigation," in *CSMR/WCRE Tools Track*, 2014.

[20] R. Metzger, *Debugging by Thinking - A Multidisciplinary approach*. Elsevier Digital Press, 2003.

[21] C. Hofer, M. Denker, and S. Ducasse, "Design and Implementation of a Backward-in-Time Debugger," in *NODE*, 2006, pp. 17–32.

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

[23] G. Pothier, E. Tanter, and J. Piquer, "Scalable Omniscient Debugging," in *OOPSLA*, 2007, pp. 535–552.

[24] G. Pothier and E. Tanter, "Summarized Trace Indexing and Querying for Scalable Back-in-Time Debugging," in *ECOOP*, 2011, pp. 558–582.

[25] A. Lienhard, T. Gîrba, and O. Nierstrasz, "Practical Object-Oriented Back-in-Time Debugging," in *ECOOP*, 2008, pp. 592–615.

[26] M. Ernst, A. Czeisler, W. Griswold, and D. Notkin, "Quickly Detecting Relevant Program Invariants," in *ICSE*, 2000, pp. 449–458.

[27] S. Hangal and M. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," in *ICSE*, 2002, pp. 291–301.

[28] R. Abreu, A. González, P. Zoeteweij, and A. van Gemund, "Automatic Software Fault Localization Using Generic Program Invariants," in *SAC*, 2008, pp. 712–717.

[29] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. Ernst, and M. Rinard, "Automatically Patching Errors in Deployed Software," in *SOSP*, 2009, pp. 87–102.

[30] N. Polikarpova, I. Ciupa, and B. Meyer, "A Comparative Study of Programmer-written and Automatically Inferred Contracts," in *ISSTA*, 2009, pp. 93–104.

[31] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Interactive Fault Localization Leveraging Simple User Feedback," in *ICSM*, 2012, pp. 67–76.