Querypoint Debugging
(Semi-Automated Inspection of Buggy Execution)

THÈSE N^{O} 5533 (2012)

PRÉSENTÉE LE 4 OCTOBRE 2012

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE TÉLÉINFORMATIQUE
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Salman MIRGHASEMI

acceptée sur proposition du jury:

Prof. M. Odersky, président du jury
Prof. C. Petitpierre, directeur de thèse
Dr J. J. Barton, rapporteur
Dr E. Bodden, rapporteur
Prof. V. Kuncak, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2012

تقدیم به همسر مهربان و دختر نازنینم،

و پدر و مادرم که در تمام زندگی همواره پشتیبانم بوده اند.

We can only see a short distance ahead,
but we can see plenty there that needs to be done.
— Alan Turing

Science is knowledge which we understand so well
that we can teach it to a computer;
and if we don't fully understand something,
it is an **art** to deal with it....
We should continually be striving
to transform every **art** into a **science**:
in the process, we advance the **art**....
Debugging is an **art** that needs much further study.
— Donald E. Knuth
The Art of Computer Programming I.

Abstract

Debugging is a hard and time-consuming programming task that appears in the most stages of software development and maintenance. Therefore, any improvement in the debugging practice can significantly reduce the time and costs of software production. In this dissertation, our focus is on *locating defects*, a major debugging task. We first analyze the widely-used and new approaches to locating-defects. From this analysis we identify several main problems in these approaches that cause accidental difficulty.

In order to better understand the locating-defects process, we systematically analyze this process based on modeling the developer's mental model. From this analysis, we develop a model that explains how a developer progresses in this stage : a developer's general strategy is to reduce the *suspicious interval* on the buggy execution until locating a *fault*. There are three basic types of movement : *controlled forward navigation*, *search*, and *causality backward navigation*.

We propose *Querypoint Debugging* as a new approach to debugging that attempts to abstract away a developer's inspection from one particular execution. We show how querypoints are defined for various types of movement, how they are processed, and how they solve most issues existing in the available debugging approaches.

We have built two prototypes of the querypoint idea. The first prototype is a general querypoint debugger for Java which is supposed to support various types of querypoints. We demonstrate how a buggy painting application can be debugged using this debugger. The second prototype is one querypoint, *lastChange*, that is integrated to a famous JavaScript debugger, Firebug. With this prototype, we show that querypoints can be smoothly integrated with traditional debuggers. Moreover, the idea can be implemented even for weakly-typed languages such as JavaScript.

We end this dissertation with providing a solution to a special but an important problem which particularly appears in JavaScript debuggers. *Anonymous* JavaScript functions are a major challenge for tool developers. After analyzing and classifying anonymous JavaScript functions in 10 large projects, we proposed an efficient algorithm that can successfully name 99 percent of anonymous functions.

Keywords: Debugging, Locating-Defects, Breakpoint, Watchpoint, Querypoint, lastChange, JavaScript, Anonymous Functions

Résumé

Le débogage est une tâche de programmation difficile et qui prend du temps. Elle apparaît dans la plupart des stades du développement de logiciels et de la maintenance. Par conséquent, toute amélioration dans la pratique du débogage peut réduire considérablement les coûts de production de logiciels. Dans cette thèse, l'accent est mis sur la localisation des défauts, une tâche majeure du débogage. Nous analysons d'abord les approches actuelles les plus utilisées et les méthodes de localisation de défauts. A partir de cette analyse, nous identifions plusieurs problèmes majeurs de ces approches.

Afin de mieux comprendre le processus de localisation des défauts, nous avons systématiquement analysé le processus en nous reposant sur la modélisation du modèle mental du développeur. De cette analyse, nous développons un modèle qui explique comment un développeur progresse dans la mise au point d'applications de l'informatique : une stratégie de développement est en général la réduction par étapes de la zone de suspicion (dans laquelle l'erreur pourrait se trouver) jusqu'à la localisation de la faute.

Il existe trois types de décision de base : la navigation « en avant », la recherche, et la navigation de causalité arrière. Nous proposons la méthode Querypoint comme nouvelle approche pour le débogage, méthode qui construit une abstraction d'une exécution particulière et qui permet ainsi à un développeur d'inspecter le code de façon optimale. Nous montrons comment des querypoints sont définis pour différents types de décision, comment ils sont traités, comment ils résolvent la plupart des problèmes qui existent dans les approches traditionnelles ou nouvelles de débogage.

Nous avons construit deux prototypes dérivés de l'idée Querypoint. Le premier prototype est un débogueur général pour Java, qui est censé soutenir différents types de querypoint. Nous montrons comment une application de dessin peut être déboguée en utilisant ce débogueur. Le deuxième prototype met en évidence lastChange, qui est intégré à un débogueur bien connu de JavaScript, Firebug. Avec ce prototype, nous montrons que Querypoint peut être facilement intégré à des débogueurs traditionnels. Par ailleurs, ce prototype montre que l'idée peut être appliquée même pour les langages faiblement typés tels que JavaScript.

Nous terminons cette thèse en fournissant une solution à un problème particulier, mais important, qui apparaît en particulier dans les débogueurs JavaScript : l'attribution de noms artificiels aux fonctions anonymes. Les fonctions anonymes de JavaScript sont un défi majeur pour les développeurs d'outils. Après l'analyse et la classification des fonctions JavaScript anonymes de 10 grands projets, nous avons proposé un algorithme efficace qui réussit à nommer 99 pour cent des fonctions anonymes.

Acknowledgements

I would like to thank my advisor, Claude Petitpierre, who gave me the freedom to choose my thesis topic, and then supported me in all stages of my journey until I could successfully defend my thesis.

I am extremely indebted to John J. Barton, my main collaborator. I met John at OOPSLA 2009 for the first time, and from then on, I have been lucky to benefit from his advice. In particular, during my three months internship at IBM Almaden, he was a great mentor when I was working on a new querypoint prototype. I believe that I could not have finished this dissertation without his support and guidance.

I want to take on this opportunity to thank my thesis committee members—Claude Petitpierre, Martin Odersky, John J. Barton, Viktor Kuncak, and Eric Bodden—who have accommodated my timing constraints despite their full schedules, and provided me with precious feedback for the presentation of the results, in both written and oral forms.

I also would like to thank my friends in the LTI group, Olivier Buchwalder and Paul-Louis Meylon, and other friends in DSLab and NSL groups for the great time and fruitful discussions we have had together.

I am grateful to professor George Candea for organizing “Advanced Topics” discussion courses in “Software” and “Operating” systems, which helped me a lot in understanding the open problems in the field.

Living in Lausanne without my good friends would not have been easy. I want to thank all of them. In particular, Armin Tajalli, Paris Jafar, Ali Rad, Maryam Zaheri, Ali Hormati, Naser Khosropour, Alireza Khadivi, Hesam Salavati, Vahid Madjidzadeh, Hossein Afshari, Mahdi Jafari, Maysam Yabandeh, Hadi Afshar, Alireza Roshanghias, Shahram Khazaei, Omid Talebi, Hossein Hojjat, Arash Amini. I am sure I have missed many others!

Every dissertation is a joint effort and mine is no exception. My beloved wife, Shahrzad, and my daughter, Nikita, deserve special thanks, because without their love and support I could not have finished my Ph.D.

Last but not least, I would like to express my love and gratitude to my parents, Mahmoud and Marzieh, for love and their support throughout the years.

Lausanne, 23 September 2012 Salman Mirghasemi

Contents

List of Figures

List of Tables

As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. **Debugging** had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

— Maurice Wilkes (1949)

1 Introduction

1.1 The Debugging Challenge

Our life is becoming more and more dependent on computers. However, computers are of no use without programs running on them. Developing new programs and maintaining legacy software costs hundreds of billions of dollars every year[1]. Moreover, building reliable, bug-free software is difficult and time consuming. Hence, identifying and facilitating the common challenging activities during software development can significantly reduce the required effort, development time, and, therefore, costs. One of the indispensable and laborious activities during software development is debugging. Debugging appears as a primary activity in almost every stage of software development (Figure 1.1).

1.1.1 A Primary Software Development Activity

Debugging is an inevitable part of programming. It is hard—or perhaps impossible—to imagine writing code without debugging. To get a piece of code right, developers need to go through the repeated cycles of *edit*, *check*[2] and *debug*. After editing code, a developer checks the program correctness by running the program. If any issues are observed during the program execution, the developer examines the source code and program states during the execution to understand the causes. Once the causes are understood, the developer tries to remove the defects and checks the program again. The process continues until no more problems are found. Then, the developer can move on and work on the next piece of code. It is a common story happens every day, not only for novice and mediocre developers, but also for very experienced ones[3].

[1]According to *datamonitor.com* report on global software industry, the software market was more than 265 billion dollars in 2010.

[2]We intentionally used *check* instead of *test* here, because this step often consists of few basic tests, which differ from a complete test suite run.

[3]Experienced developers perform better than other developers because they usually need less time to finish a step and fewer cycles to accomplish the whole task.

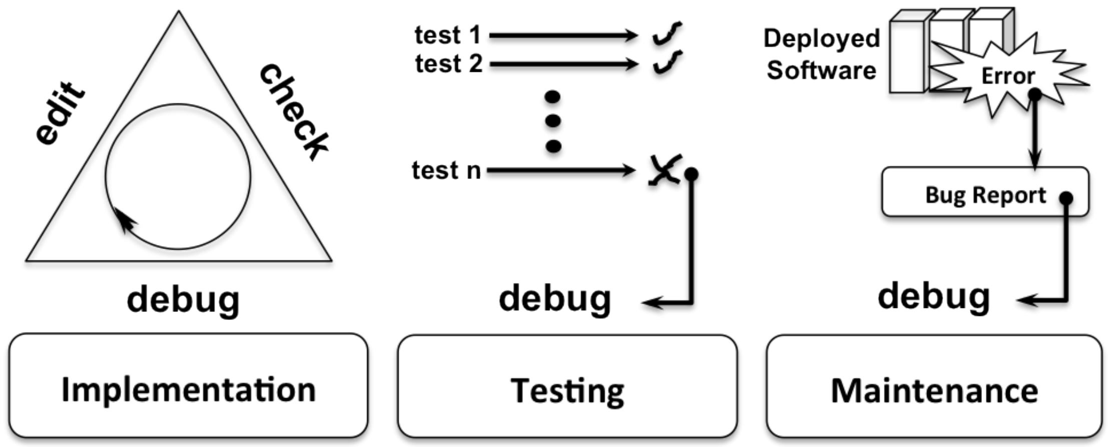


Figure 1.1: Debugging is a primary activity during implementation, testing, and maintenance.

Debugging is the complement of testing. Lightweight checks during development cycles are not sufficient to catch all bugs and build a robust, bug-free software. Every new version of program must go through a rigorous testing process. When a new bug is discovered during this process, it has still to be fixed by developers. Developers reproduce the bug employing the test cases that disclosed the bug for the first time. Then, they debug to fix the bug.

Debugging is also a basic task during program maintenance and support [Vessey1985]. Despite all quality controls, it often happens that bugs show up on the users' machines. These bugs are reported, either by users or automated feedback mechanisms, to the development team. Developers have to reproduce the bugs by analyzing these reports, debug the program and provide patches or new versions of program to the affected users as soon as possible.

A significant part of debugging consists of program comprehension. Developers need to understand and explain how defects eventually cause the bug symptoms before they can fix the bug. Hence, many debugging techniques and tools are mainly developed for program comprehension. According to [LaToza2007], using debuggers, after reading source code, is the most common approach for code understanding among developers. On the other hand, program comprehension is a basic activity for most development tasks. Therefore, improving debugging tools usually reduces the time and difficulty of other development activities as well.

1.1.2 Hard and Time Consuming

Debugging is notoriously difficult and extremely time consuming. In order to fix a bug, developers typically reproduce and monitor the buggy execution several times to understand the program's unexpected behavior. Trial-and-error, guess-work, and analyzing complicated data structures are very common tasks during debugging. Brian Kernighan, one of the creators of Unix operating system, once said: “Debugging is *twice as hard as* writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

The above argument attempts to ironically tell us how difficult debugging is, however, it also touches another important fact; when developers write code, they have a design or blueprint that helps them to go through a path until they achieve the result, however, during debugging, they don't have any idea about the next step before knowing the outcome of the current step. They have to update their plan and devise a new strategy at every step based on the new findings.

In other words, debugging is the art of walking on the narrow uncertain rope of assumptions in the dark space of unknowns. It starts with a few apparent bug symptoms and continues with a long sequence of hard-to-answer questions: Do these bug symptoms really represent a bug?, If the answer is no, what is the explanation?, Otherwise, what are the defects that must be fixed?, How can I reproduce the bug?, What is the expected program state at this execution point?, Is the program state is exactly as it is supposed to be?, How did this runtime state occur?, I don't have the source code for this suspicious library, how can I check that it is not responsible for the bug?, There is a complex data structure which is evolved during program execution, how can I check its correctness somewhere at the middle of execution?

The above list is just a small subset of possible questions that come to the mind of developers during the process of debugging[4]. Answering each question can take a significant time. This partially explains why developers spend about fifty percent of their time debugging [LaToza2006] and the average error takes 17.4 hours to find and fix [Tassey2002]. Moreover, in most cases developers are not fully confident about their answers since they had to make several assumptions before finding the answer. Unfortunately, these unchecked assumptions cause debugging to take even more time [Mayrhauser1997].

1.1.3 More Bugs to Fix

The debugging techniques and tools remained pretty much similar to the early days of software development, whereas the size and complexity of today's software is tremendously greater than those day's. Every day, millions of software developers work to improve the world's software infrastructure but must do so with little more than a breakpoint and a print statement.

Although developers spend a huge amount of their time on debugging, at the end of the day, they do not have any tangible artifact for reuse. The implicit knowledge about code and hopefully one more fixed bug are all outcomes of this difficult process. If the same bug is again reported a few weeks later, the developer has to go through the same steps without being able to reuse the previous inspections.

During the last decade, testing, with the goal of improving software quality and reliability, received a serious attention from both industry and academia. As a consequence, nowadays,

[4]For an interesting list of *Hard-to-Answer* questions about code refer to [LaToza2010].

developers employ sophisticated and powerful techniques and tools that enable them to reveal more bugs in a shorter time. However, to make all this progress fruitful, developers must be able to fix bugs in a faster pace.

1.2 The Scope of This Dissertation

The details of the debugging process may vary depending on the type of bug, where, when, and how the bug happens, nonetheless it often contains four main stages: reproducing the bug, simplifying the bug, locating defects that cause the bug, and fixing defects (Figure 1.2).

The first stage is *bug reproduction*. The goal of this phase is creating an instance of program execution that produces the same bug symptoms. The output of this stage is often an initial configuration and a test case which includes input data, necessary events and actions to reproduce the bug symptoms.

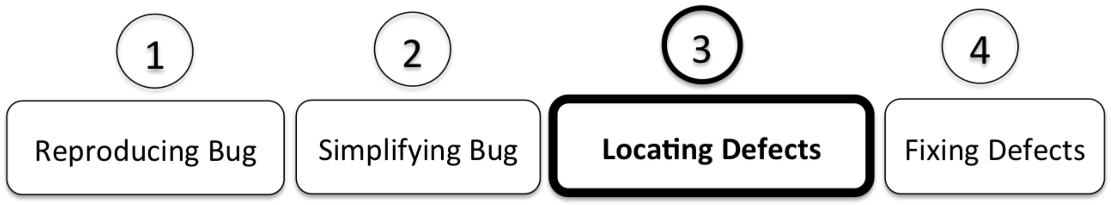


Figure 1.2: The four phases of debugging: reproducing bug, simplifying bug, locating defects and fixing defects. The third stage, *locating defects*, is the focus of this dissertation.

There are two main sources of difficulty at this stage. The first issue is the lack of enough data about the initial configuration and actions that eventually produce bug symptoms. This issue particularly appears when developers have to reproduce bugs from bug reports they receive from users' machines.

The second issue is non-determinism. A bug is called non-deterministic by developers when starting from the initial state and applying the same actions not always reproduces the bug symptoms. Although the non-deterministic adjective comes with bugs, it describes the inability of developers to deterministically reproduce the bug. Non-deterministic bugs often depend on low-level or hidden events that are hard to identify and set appropriately by the developer.

The second stage of debugging is bug simplification. The goal of this phase is to create a smaller test case that reproduces the bug symptoms. In order to achieve this goal, developers usually try to recognize and remove unnecessary data, events and actions for the bug reproduction. The test case simplification can significantly decrease the total number of events in the buggy execution and the time needed before the bug symptoms appear. Moreover, it facilitates the next stage, locating defects, by shortening the size of search space.

Locating defects that cause the bug, along with providing an explanation on how these defects eventually cause the bug symptoms, are at the center of the debugging process. Moreover, most of the debugging time is usually spent at this stage [Eisenstadt1997]. The common approach in this stage is inspecting the buggy execution. Developers compare at different execution points the program state with the initial expectations to understand where and why they begin to diverge.

The last step of debugging is fixing defects. Although fixing the discovered defects is usually straightforward [Eisenstadt1997], developers must be careful about two important issues. First, the bug fix covers all cases of error [Gu2010]. Second, the bug fix itself does not introduce new bugs [Yin2011].

The focus of this dissertation is on the problems appearing during the locating defects phase. Therefore, in this dissertation, we always implicitly assume that **the bug has already become reproducible for the developer**. Note that, since these stages are independent from each other and a bug must become reproducible before a developer can start locating defects[5], this assumption does not impose an additional constraint on the process of locating defects.

Two distinct categories of approaches to locating defects are *tool support enhancement* and *automated fault localization*. The goal of approaches in the former category is to improve tools which are used by developers for locating defects, whereas the aim of approaches in the latter category is to carry out the whole task without or with very limited assistance of developers. This dissertation belongs to the first category.

1.3 Problem Statement

Inspecting the buggy execution, which is the common approach to locating defects, is hard and time-consuming. The key question of this dissertation is the following :

Is it possible to reduce the effort and time needed for inspecting the buggy execution, and therefore locating defects, by improving the current debugging approaches and enhancing tool support? How?

There are two widely-used approaches to inspecting an execution : *log-based debugging* and *breakpoint-based debugging*. In the first approach, a developer inserts logging statements within the source code, in order to produce an ad-hoc trace during program execution. In the second approach a developer runs the program under a dedicated debugger which allows halting the execution at determined points, called *breakpoints*, inspecting memory contents, and then monitoring the execution step-by-step.

Unfortunately, these approaches suffer from several issues. In chapter 2 we enumerate and discuss the problems in details. Among them are the follwoing :

[5]Because locating defects is mainly carried out by inspecting the buggy execution. Otherwise, it is reduced to reading and verifying the source code.

1. Both approaches require a lot of guesswork; developers have to guess the locations of log statements and breakpoins by browsing and searching through source files.
2. Developers have to analyze and memorize a lot of data before they could obtain a useful piece of information.
3. Moving backward, in other words obtaining the program state of an earlier point on an execution, is complicated or even impossible.
4. None of the approaches produces debugging artifacts, which can be used for restoring the debugging state if it is needed later, e.g., if a fixed bug is reported again.

Several new approaches, as we discuss in chapter 2, have been developed and proposed in order to mitigate some of the above problems. Although these proposals are able to partially or totally overcome some of these issues, they have their own limitations and problems. For example, *instruction indexing on identical re-executions* and *capture-replay*, two different approaches introduced for supporting backward navigation, incur too much execution overhead, 10 to 300 times of the original execution, which makes using them impractical for developers.

As a result, none of these approaches could widely replace or significantly improve the traditional approaches employed by developers for many years. We can name two main challenges in this regard; First, most of the proposed approaches suffer from scalability issues. Second, they are not flexible enough and cannot be easily integrated with the traditional tools and techniques used by developers.

1.4 This Thesis

We pursue a totally different approach to tackle debugging issues. *Abstraction* and *reuse*, two basic concepts that have brought about many significant improvements in software engineering, are at the core of our approach. We also value *flexiblity* and *ease-of-use* in our proposal, **Querypoint Debugging**.

Querypoint debugging is based on the *theory of locating-defects* presented in chapter 3. In this theory, the locating-defects process is modeled as a sequence of execution point inspections with the aim of reducing the length of *suspicious interval* until the *fault* is identified. In this model, three basic types of execution point selection are described : *Controlled forward stepping, casuality backward navigation* and *search.*

A *querypoint* abstracts away an execution point from a particular execution as a developer does. A querypoint usually contains *how* or *why* an execution point is inspected, the *values* that are inspected and *the outcome* of inspection. This concept not only allows us to support atomic actions which are mapped from atomic user goals, as David Ungar suggests [Eisenstadt1997], but also to provide efficient algorithms to perform these actions. Querypoints also enable us to create reusable debugging artifacts and check the consistency of developer's actions.

Querypoints can smoothly be integrated to the traditional approaches. As we demonstrate in chapter 6, *lastChange*, which is a querypoint that selects the last write to a memory location, is integrated to a breakpoint-based debugger. If a developer is suspicious about a particular value, for example when they are inspecting the program state at a breakpoint hit, they can use *lastChange* to obtain the program state at the execution point where the value was set. The debugger re-executes the buggy program, collects very limited data and finally identifies and shows the requested program state to the developer.

Querypoints facilitate navigation and data collection over the buggy execution. However, a developer has still to understand the program state at different execution points. A particular problem that appears in JavaScript debuggers is due to *anonymous* functions. Although developers can see the program state in the debugger, they can not understand it because of many anonymous items. In chapter 7, as a solution to this problem, we propose a new algorithm, *nonymous*, which constructs meaningful identifiers for these anonymous items.

This work evaluates the thesis statement presented in this section, along with a number of smaller, related hypotheses.

1.4.1 Thesis Statement

Understanding how developers inspect and percept a buggy execution opens new doors to improving the debugging practice :

Querypoints, which encapsulate a developer's inspections on the buggy execution based on strategy, purpose and reason, provide a basis for supporting a higher level of abstraction in debuggers, which not only facilitates the locating-defects process, but addresses many debugging issues such as the difficulty of backward navigation and the lack of debugging artifacts, and enables debuggers to restore and verify the debugging state on a new instance of buggy execution.

Anonymous or nameless objects in a program state (such as anonymous functions in the JavaScript language) can be efficiently named using an algorithm which emulates the approach developers use for describing these nameless objects.

1.4.2 Hypotheses

We can break the thesis statement down into more concrete, and measurable hypotheses.

Hypothesis 1

Developers locate defects by inspecting the buggy execution and identifying faults. Execution points play a key role in this process. The whole process can be modeled as a sequence of execution point inspections with the aim of reducing the length of *suspicious interval.* A developer selects the subsequent execution point in one of the following general ways :

controlled forward stepping, casuality backward navigation and *search*.

Validation : The above hypothesis is the result of a systematic analysis of the locating-defects process in chapter 3. It conforms with previous debugging and program comprehension theories. As demonstrated in chapter 3, using this theory, we can evaluate new debugging tools, identify the key directions for enhancing debuggers and the challenges in this path.

Hypothesis 2

Querypoints can abstract away execution points from executions as a developer does. We can summarize the whole buggy execution inspection by querypoints. They provide solutions to most problems that exist in the traditional debugging approaches.

Validation : In chapter 4, we define and explain different types of querypoints. We demonstrate that for every type of execution inspection a querypoint can be defined and these querypoints can effectively solve most problems listed in chapter 2.

Hypothesis 3

Querypoint debuggers are feasible and can be integrated to traditional debuggers.

Validation : In chapter 5, we explain the architecture of a querypoint debugger for the Java language. In chapter 6, we describe the details of implementation of *lastChange* querypoint for JavaScript language and its integration with a breakpoint-based debugger.

Hypothesis 4

Most JavaScript function are anonymous, which causes a serious comprehension issue in JavaScript editors, debuggers and profilers. It is possible to automatically provide meaningful names for most of these functions.

Validation : In chapter 7, we first analyze 10 large JavaScript projects and show that more that 93 percent of JavaScript functions are anonymous. We provide an algorithm which is able to construct practical meaningful names for more that 99 percent of these functions.

1.5 Contributions

This thesis makes the following contributions:

1. Analysis and comparison of widely-used and new debugging approaches and tools (Chapter 2).

2. A new comprehensive model of locating-defects process, which can be used for analyzing and evaluating debugging approaches and tools (Chapter 3).
3. The *querypoint* concept and its variations for *controlled forward stepping, causality backward navigation* and *search* (Chapter 4).
4. A querypoint debugger for Java based on a generic trace query language (Chapter 5).
5. Integrating *lastChange* querypoint to Firebug, a breakpoint-based JavaScript debugger.
6. Analyzing and classifying anonymous functions usage in 10 large JavaScript Projects (Chapter 7).
7. The *nonymous* algorithm for naming anonymous JavaScript functions (Chapter 7).

1.6 Impact

The *purple* project[6] is an experimental querypoint debugger which is being developed at Google. The *re-run* button[7] in the Firebug JavaScript debugger was first developed for the querypoint plug-in in order to provide shorter re-execution cycles. This capability is added and remains an important feature of Firebug.

Our research on anonymous JavaScript functions is cited by the EcmaScript committee[8] and the Firebug team[9]. A reference implementation of Nonymous alogrithm is developed[10] and is used in the Orion Javascript editor for providing an outliner[11]. The algorithm is implemented in Firefox and will be included in the Firefox release (16.0)[12].

1.7 Structure of this Dissertation

The rest of this dissertation is structured as follows. In chapter 2, we describe, analyze and discuss various debugging approaches and tools. Chapter 3 explains the theory of locating-defects. We explain the *querypoint debugging* in chapter 4. The next chapter discusses the design and architecture of a querypoint debugger. Chapter 6 includes the details of implementing an important querypoint, *lastChange*, and its integration to breakpoint-based debugging. We discuss user interface inspection at different abstraction levels in chapter 7. *Nonymous* algorithm is presented in chapter 7. Finally, we conclude in chapter 8.

[6]https://github.com/johnjbarton/Purple
[7]http://blog.getfirebug.com/2010/08/31/firebug-1-7a1
[8]https://bugzilla.mozilla.org/show_bug.cgi?id=433529
[9]http://blog.getfirebug.com/2011/04/28/naming-anonymous-javascript-functions
[10]https://github.com/johnjbarton/nonymous
[11]http://planetorion.org/news/2011/09/orion-0-3-m2-new-and-noteworthy
[12]http://www.mozilla.org/en-US/firefox/16.0/releasenotes

It is good to rub and polish our brain
against that of others.
— Michel de Montaigne(1533-1592)

2 Related Work

Various debugging approaches, tools and techniques have been introduced in the programming history, though only a few of them, which we call *widely-used*[1] debugging approaches, are being broadly used by software developers. In this chapter we will enumerate the major debugging approaches, analyze and compare them in order to obtain a high-level picture of the current state of debugging.

We first briefly describe and discuss the widely-used debugging approaches. We then discuss new debugging approaches and tools. For each approach or tool we answer the following questions:

1. What are the problems that the approach/tool attempts to address?
2. What are the features that the approach/tool provides?
3. How does the approach/tool differ from other approaches/tools?
4. What are the problems/limitations of the approach/tool?
5. What makes the widely-used methods advantageous? What are the barriers limiting the use of these new approaches/tools?

Finally, we provide a list of potential improvements to widely-used debugging approaches and use this list to classify new debugging approaches.

2.1 Widely-Used Approaches

The two widely-used methods for inspecting a buggy execution are *log-based* debugging and *breakpoint-based* debugging.

[1] They are also called *traditional* debugging approaches.

2.1.1 Log-based Debugging

Log-based debugging is the simplest and most accessible debugging technique. It does not require any special external tool (e.g., a debugger) for inspecting the internal program states. A developer can insert log statements within the source code, in order to produce an ad-hoc trace during program execution. A log statement is itself an instruction, or a set of instructions, that read(s) and record(s) values from the program state. Therefore, whenever a log statement is executed, it records parts of the program state, which can be inspected later by a developer. For example, in order to inspect values passed as parameters to a function, a developer can insert a log statement that records the parameters at the beginning of the function block.

Log-based debugging suffers, however, from several problems. A developer, in order to obtain the program state at a suspicious execution point, has to speculate about the *location* where the log statement must be inserted and the *data* that must be collected. Usually a developer has to modify a log statement several times due to insufficient collected data, or wait a long time because too much data are recorded. Once adequate data are collected, analysis is still required to understand it. Developers usually end up dealing with long trace files and analyzing huge amounts of collected data. In other words, the technique can expose the actual history of execution but it requires cumbersome and widespread modifications to the source code; and it does not scale, because manual analysis of huge traces is hard.

In order to mitigate the scalability problem different mechanisms are employed . A simple mechanism is to split a huge trace file into a number of smaller files, each one containing the trace of only one aspect of the execution. In a different mechanism, a developer can set a *severity level* for a log statement, so the log statement is effective only if the *trace level*, which is again set by the developer before the execution, is not greater than the severity level of the log statement. Another mechanism allows a developer to group log statements in hierarchies. Therefore the developer can set the trace level of each node in the hierarchy separately[2]. Although these configuration mechanisms make the scalability problem slightly less painful, they do not solve it. Moreover, they are not convenient to use, and in practice, they do not provide the flexibility needed for inspecting narrow intervals of execution.

2.1.2 Breakpoint-based Debugging

Breakpoint-based debugging consists in running the program under a dedicated debugger, which allows a developer to pause the execution at determined points, called *breakpoints*, to inspect the program state, and then to continue the execution step by step. A breakpoint, in its simplest form of a *line* breakpoint, is set on an instruction. When the program is executed, it halts before the instruction execution—a breakpoint is hit or a breakpoint occurs. At a breakpoint hit, a developer can inspect the program internal state with the help of the debugger. In other words, similar to a log statement, a breakpoint allows a developer to inspect a program state; but unlike a log statement, it does not require any modification of the source

[2]For example, *log4j* (http://logging.apache.org/log4j) supports all these features.

code.

The idea behind line breakpoints can be extended. If an instruction execution is considered an event, a breakpoint selects certain events in an execution. By choosing different event types, new forms of breakpoints can be introduced. For example, a *method* breakpoint, set on a method, halts an execution whenever the method, or one of its overwriting methods, is called. Similarly, an *exception* breakpoint, set on an `Exception` class, halts an execution whenever an instance of that exception type, or one of its subtypes, is thrown. A *watchpoint* is set on a property or variable. It functions like a breakpoint that is set on the instructions that access or set the property or variable.

At a breakpoint hit, a developer can access almost any part of the program state, and therefore does not need to decide about data to be collected, unlike the log-based approach, when to set a breakpoint. However, identifying the location where the breakpoint must be inserted, similar to the log-based approach, entails programmer knowledge and expertise. Breakpoints are mainly effective in forward debugging. But, unlike log-based debugging, the information about the previous state and the activity of the program is limited to the inspection of the current call stack. In order to overcome this limitation, some debuggers support *rewinding* the execution from a former instruction in the call stack[3] . This feature does not roll back the program state and only works in cases where the rolled-back instructions have not changed global variables or did not have any side effect.

A developer has to resume the program execution after every breakpoint hit. As the number of breakpoint hits increases, the process of inspecting the program state, collecting data and resuming the execution becomes more and more cumbersome. *Conditional* breakpoints are a solution for reducing the number of irrelevant breakpoint hits. A developer can set a *hit-count* for a breakpoint. In this case, the debugger only halts the execution after the breakpoint has been hit hit-count times. A developer can also set a condition on the values of the program state at the breakpoint. The execution halts only if the condition holds when the breakpoint is hit. Unfortunately, conditional breakpoints are neither effective enough nor practical enough to fulfill developers' needs in debugging [Bodden2011].

2.2 New Debugging Approaches

As we mentioned at the beginning of this chapter, many debugging tools and techniques have been introduced in order to improve the debugging experience. We identify a few key ideas that are used in these tools. In this section we will discuss these key ideas, the problems they attack and their limitations.

[3]For example, *Eclipse*'s java debugger supports this feature by its *drop to frame* command.

2.2.1 Instruction/Event Indexing on Identical Re-Executions

This approach, as its name suggests, is based on reproducing identical traces of the original execution : In every re-execution all instructions must be executed in the same order as the original execution, and the same sequence of program states must be created. In other words, the same input must be applied in the re-execution, and there should be no sources of non-determinism.

The main purpose of this approach is to facilitate backward navigation. Suppose that an *index* is assigned to every instruction in an execution, for example using a *counter* that is increased after every instruction execution. Thus, moving one step backward from an instruction with index *i* corresponds to moving to the instruction with index *i-1*. Because a re-execution is an identical copy of the original execution, every instruction will receive the same index. Therefore, for backward movement to an instruction with a lower index *j*, it is enough to re-execute and halt the execution at the *jth* instruction.

Identical re-execution is usually accompanied with a *checkpointing* feature : It stores a snapshot of program state, called *checkpoint*, in time intervals. The program can be re-executed from a checkpoint, for example the last one, which allows for faster re-executions. This feature can significantly reduce the debugging time in the case of long executions. Another advantage of this approach over widely-used approaches is its ability to restore the debugging state after re-execution; as executions are identical, the developer's last inspection can be re-located on the re-execution using its index.

This approach, however, has several limitations. One issue is execution slowdown. An initial execution is slower than the original execution due to storing non-deterministic events and checkpointing. Re-execution is also slower than normal re-execution because of re-applying non-deterministic events and assigning indices to instructions. The execution overhead can vary from 15 to 68 times, depending on the implementation details and program type [Boothe2000, Maruyama2003, Bhansali2006].

Another important barrier to employing this approach, in addition to execution overhead, is the requirement for identical, instruction by instruction, re-executions. In fact, even one instruction difference between two executions can lead to wrong results. Although the research on deterministic execution replay is very active, fulfilling this requirement in real-world software development, which usually deals with different inputs and sources of non-determinism, is still a challenge and requires much effort.

2.2.2 Capture-Replay

Capture-Replay attacks similar problems that the previous approach have tried to resolve. The key idea is simple : If enough data about various aspects of execution are captured, then there is no need for re-execution; because any program state in the execution can be restored. As a result, the problems in the widely-used approaches due to re-execution do not exist here.

Moreover, it also addresses backward navigation. In fact, moving backward on the collected trace is similar to moving forward in this approach.

But, Capture-Replay suffers from different issues. First, the recording phase is time expensive—from 10 to 300 times the original execution time [Pothier2011]—and it should be repeated in the case of changes in the program. Second, the execution trace cannot fully replace the live execution. There are other aspects of execution (e.g., program user interface, file system, and database tables) which are also important in debugging and are not available to the developer in this approach. Third, querying collected data (e.g., to restore the program state at a certain point) might not be efficient enough for debugging realistic programs.

2.2.3 Query-Based

An approach to eliminating or reducing guesswork is to provide expressive querying means that allow developers to express easily what they are looking for with little effort. A query selects one or a set of program states or events in the execution : for example, a log statement, a breakpoint or a step-over are all simple types of a query. Moreover, query-based debugging tools can also reduce the amount of irrelevant data that a developer has to deal with.

A query can be processed on-line, on a live execution, or offline on the execution trace, depending on the navigation mechanism used in the tool. We can identify three main challenges for this approach. First, processing a query should not take a long time. Second, a querying mechanism should be easy to use; for example, using predefined queries in a debugging tool is much more convenient than queries that must be defined first by a developer in a particular language. Third, because one type of query does not usually fulfill all developers' needs, query-based tools should be easily integrated with other debugging approaches.

2.2.4 Program Slicing

Program slicing [Weiser1979] is the computation of the set of program statements, the *program slice*, that could affect the values at some point of interest, referred to as a *slicing criterion*. It is a general technique with a variety of applications including debugging. Program slicing can be used in debugging to reduce the search space for a developer by ignoring irrelevant program statements. Program slicing can be considered as a special form of query-based approach, where a slicing criterion is a query and its corresponding program slice is the result of the query.

Various slightly different notions of program slices have been proposed, as well as a number of methods to compute them. An important distinction is between a *static* and a *dynamic* slice. A *static* program slice S consists of all statements in program P that could affect the value of variable v at some point p. A *dynamic* slice contains all statements that actually affect the value of a variable at a program point for a particular execution of the program rather than all statements that could affect the value of a variable at a program point for any arbitrary execution of the program [Tip1995].

In order for program slicing to be effective in debugging, the slice size should not be large. This is the main challenge of program slicing. Another challenge is to compute slices efficiently. For example, in the case of dynamic slicing, a complete trace of execution is required before the slicing algorithms can be applied.

2.2.5 Algorithmic Debugging

Algorithmic debugging [Shapiro1982] tackles a different set of problems. It partially automates the task of localizing a bug by comparing the *intended* program behavior with the *actual* program behavior. The intended behavior is obtained by asking the user whether or not a program unit (e.g., a procedure) behaves correctly. By using these responses, the location of the bug can be determined at the unit level.

An algorithmic debugger first builds the execution trace at the procedure/function level. Each node in the tree corresponds to an invocation of a procedure or function and holds a record of supplied arguments and returned results. Once built, the debugger basically traverses the tree in a top-down manner, asking, for each encountered node, whether the recorded procedure or function invocation is correct or not. If not, the debugger will continue with the child nodes, otherwise with the next sibling. A bug is found when an erroneous application node is identified where all children (if any) behave correctly.

Algorithmic debugging suffers from the following issues. First, capturing and visualising the whole trace is not practical. Second, for realistic programs the number of questions asked by the debugger is too large. Third, the questions asked are often complex, involving large data structures and unevaluated expressions.

2.3 New Debugging Tools

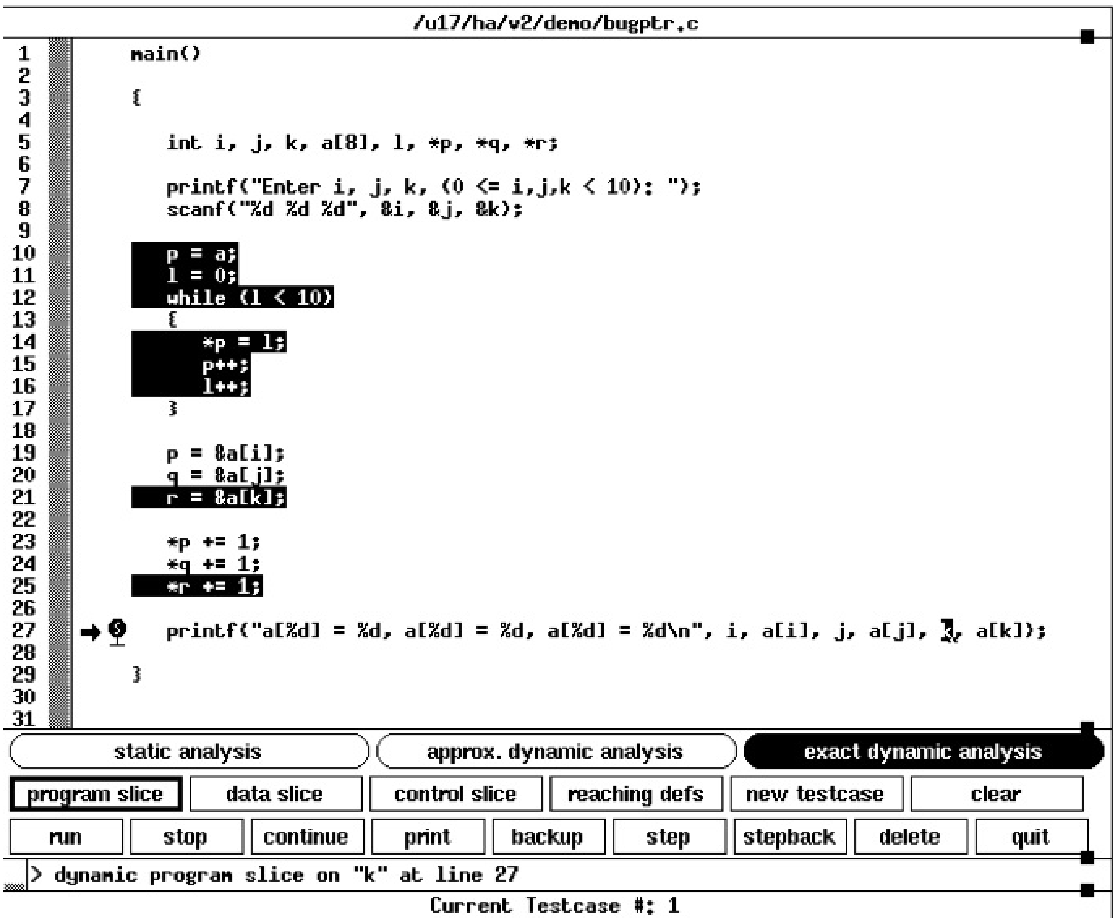
This section is a collection of distinguished debugging tools that had remarkable or interesting contributions to the debugging practice. Although most of these tools are not used by software developers, they have improved the state of the art in one or several areas. The tools are presented in chronological order, however we discuss and compare related tools. Our focus is on general debugging tools, so we do not discuss tools that, for example, only enable deterministic replay of executions and do not provide any new debugging features or tools that are used for diagnosing only one special type of bug.

2.3.1 EXDAMS [Balzer1969]

EXDAMS, to the best of our knowledge, is one of the first capture-replay debuggers built for Fortran. It stores the history of a program execution on tape and, in the replay phase, reads from the tape. In addition to supporting a weaker version of ordinary forward and backward single stepping, EXDAMS supports *motion-picture view*, which allows the developer to select a set of variables and a direction, forward or backward, in order to monitor changes in the variables' values.

2.3.2 SPYDER [Agrawal1990, Agrawal1993]

SPYDER is a capture-replay debugger that employs the idea of *dynamic slicing* [Agrawal1990]. A dynamic slice for a set of values at a given execution point comprises all statements in the history of execution that affect this set of values. In order to compute a dynamic slice, SPYDER needs the complete execution history. It then analyzes this history, along with the source code, to specify the dynamic slice. In addition to ordinary backward stepping, SPYDER can show the program slice, *data slice* and *control slice* to the developer. A *data slice* is obtained by taking only (static or dynamic) data dependences into account. A *control slice* consists of the set of (static or dynamic) control predicates surrounding a language construct. (Figure 2.1).



```
1      main()
2
3      {
4
5         int i, j, k, a[8], l, *p, *q, *r;
6
7         printf("Enter i, j, k, (0 <= i,j,k < 10): ");
8         scanf("%d %d %d", &i, &j, &k);
9
10        p = a;
11        l = 0;
12        while (l < 10)
13        {
14           *p = l;
15           p++;
16           l++;
17        }
18
19        p = &a[i];
20        q = &a[j];
21        r = &a[k];
22
23        *p += 1;
24        *q += 1;
25        *r += 1;
26
27        printf("a[%d] = %d, a[%d] = %d, a[%d] = %d\n", i, a[i], j, a[j], k, a[k]);
28
29     }
30
31
```

```
> dynamic program slice on "k" at line 27
Current Testcase #: 1
```

Figure 2.1: This screenshot shows SPYDER special features including program, data and control slice buttons in addition to regular forward and backward stepping buttons.

SPYDER assumes a deterministic execution without side-effects[4], which is a very strong assumption. It considers a program state as a set of variable values and the location of program control. As a result, every instruction execution changes either one variable value or the control location. Thus for backtracking, it is enough to undo one of these two effects.

2.3.3 bdb [Boothe2000]

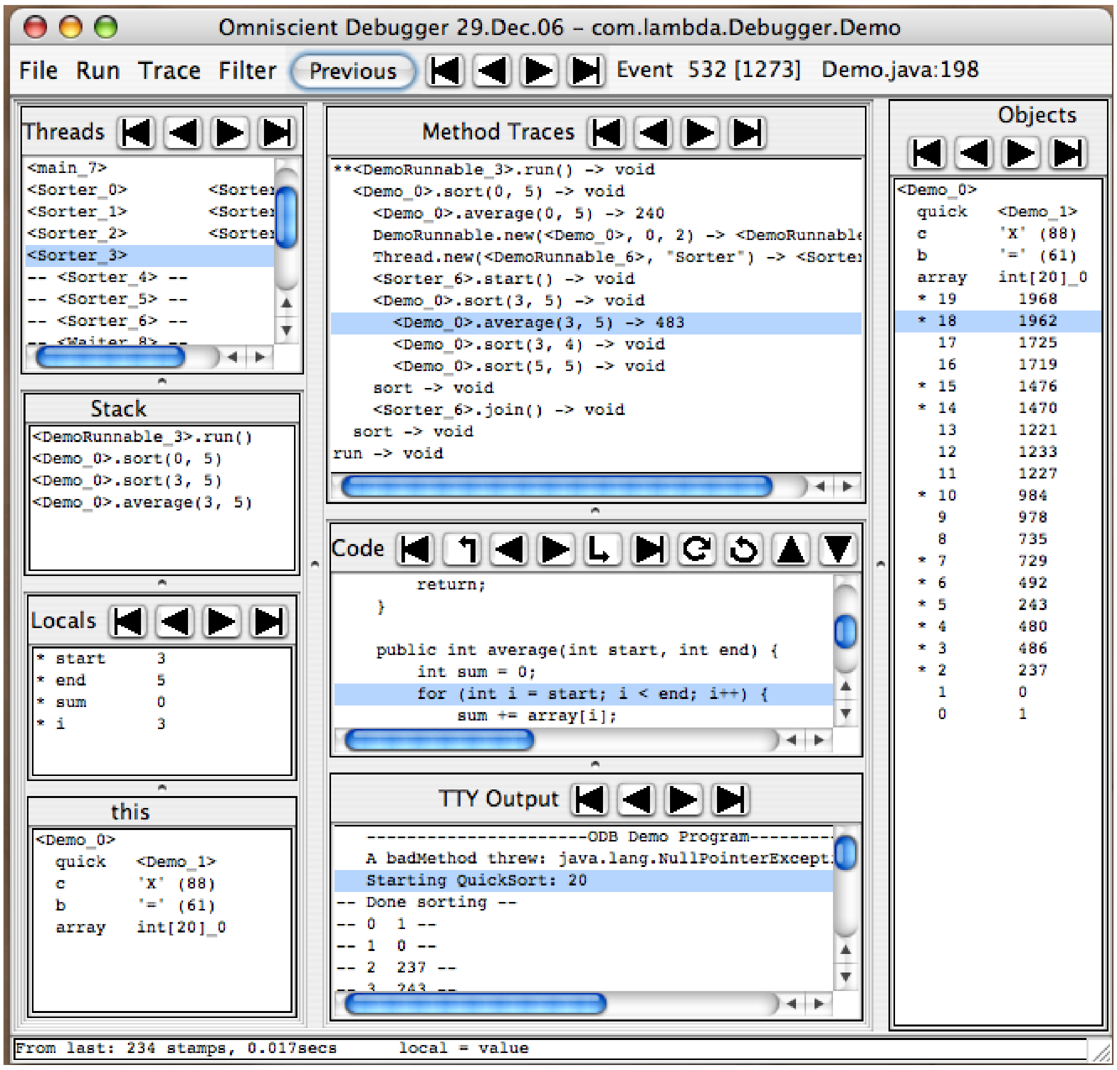
bdb is a bidirectional C debugger that employs "indexing instructions on identical re-executions" approach in order to support backward navigation. It provides various stepping features similar to those that are available in ordinary breakpoint-based debuggers, but for both directions. For example, *bstep* is a backward step, whereas *step* is an ordinary forward step[5]. Other backward movements are *bcontinue*, *previous*, *before*, and *buntil*, which are the reverse of the forward movements *continue*, *next*, *finish*, and *until*. *Continue* is similar to hit-count for breakpoints and receives an integer, which tells the debugger how many breakpoint hit must be ignored. *Next* works like step over in regular debuggers. *Until* functions as a watchpoint with a condition on a variable value.

bdb supports various stepping features by employing several counters, each one assigns indices required for one type of movement. For identical re-execution, bdb records the results of the non-deterministic system calls and re-injects them into the program when it is replayed. It also makes use of checkpoints to reduce the time needed for re-execution. In order to measure only the counters' overhead and exclude the overhead of identical re-execution, recording and re-injecting features were set off and deterministic executions were used. Counters incur the execution overhead two to four times more for both directions. Moreover, some backward movements, including *previous*, *before*, and *buntil* require two passes, i.e., in the first re-execution appropriate indices are assigned and in the second re-execution, debugger halts at the right index. The overall overhead of bdb can be 51 times of the original execution.

2.3.4 Omniscient Debugger [Lewis2003]

Omniscient debugger, ODB, is a capture-replay debugger that keeps the history of events in memory hence can only record and store the complete history for a short period of time. It improves SPYDER by supporting multiple threads and also by capturing the program standard output in addition to the program internal state. It records all the events that occur during the buggy execution and later lets the developer navigate through the obtained execution log (Figure 2.2). Unfortunately ODM incurs significant execution overhead, about 100 to 200 times.

[4]It seems that it also assumes only one active thread.
[5]*step into*



```
<main_7>
<Sorter_0>          <Sorte
<Sorter_1>          <Sorte
<Sorter_2>          <Sorte
<Sorter_3>
-- <Sorter_4> --
-- <Sorter_5> --
-- <Sorter_6> --
```

```
**<DemoRunnable_3>.run() -> void
  <Demo_0>.sort(0, 5) -> void
    <Demo_0>.average(0, 5) -> 240
    DemoRunnable.new(<Demo_0>, 0, 2) -> <DemoRunnable
    Thread.new(<DemoRunnable_6>, "Sorter") -> <Sorte
    <Sorter_6>.start() -> void
    <Demo_0>.sort(3, 5) -> void
      <Demo_0>.average(3, 5) -> 483
      <Demo_0>.sort(3, 4) -> void
      <Demo_0>.sort(5, 5) -> void
    sort -> void
    <Sorter_6>.join() -> void
  sort -> void
run -> void
```

```
<Demo_0>
  quick     <Demo_1>
  c         'X' (88)
  b         '=' (61)
  array     int[20]_0
  * 19        1968
  * 18        1962
    17        1725
    16        1719
  * 15        1476
  * 14        1470
    13        1221
    12        1233
    11        1227
  * 10        984
    9         978
    8         735
  * 7         729
  * 6         492
  * 5         243
  * 4         480
  * 3         486
  * 2         237
    1         0
    0         1
```

```
<DemoRunnable_3>.run()
<Demo_0>.sort(0, 5)
<Demo_0>.sort(3, 5)
<Demo_0>.average(3, 5)
```

```
      return;
    }

    public int average(int start, int end) {
        int sum = 0;
        for (int i = start; i < end; i++) {
            sum += array[i];
```

```
* start      3
* end        5
* sum        0
* i          3
```

```
<Demo_0>
  quick     <Demo_1>
  c         'X' (88)
  b         '=' (61)
  array     int[20]_0
```

```
  ---------------------ODB Demo Program--------
  A badMethod threw: java.lang.NullPointerExcept
  Starting QuickSort: 20
-- Done sorting --
-- 0  1 --
-- 1  0 --
-- 2  237 --
```

```
From last: 234 stamps, 0.017secs       local = value
```

Figure 2.2: An omniscient debugger screenshot, which demonstrates its bidirectional navigation feature.

2.3.5 DQBD [Lencevicius2003]

Lencevicius et al. studied supporting queries similar to conditional breakpoints that set constraints on the program state. Their debugger, *dynamic query-based debugger (DQBD)*, can efficiently check complex constraints on the objects in heap during program execution.

2.3.6 Reverse Watchpoint [Maruyama2003]

Reverse watchpoint, like bdb, employs an "indexing instructions" approach; but unlike bdb, which supports various types of stepping, it is a single feature debugger. Reverse watchpoint, as its name suggests, is the reverse of regular watchpoints. Again, it relies on identical re-execution, though it does not support it. So, it only works for deterministic one-thread executions. In order to reduce the counter overhead, it uses a slightly different index, a pair made up of a line number and a timestamp. A timestamp is an integer that is increased at method calls and loop returns, so an instruction receives different indices if it is executed more than once. Reverse watchpoint, similar to *buntil* in bdb, needs two passes and its counter incurs 1.5 to 2.5 times more execution overhead.

2.3.7 Time Travelling Debugger (TTD) Based on NIRVANA [Bhansali2006]

Nirvana is a deterministic replay system for native programs, which properly supports multi-threaded programs. It records the result of memory reads to account for scheduling-induced non-determinism. The tracing overhead of Nirvana is between 5 and 17 times more. The time to re-simulate a traced application is about 15 to 68 times the cost of native execution time. The time travelling debugger, which is built on top of Nirvana, provides features similar to bdb. No numbers are provided for the performance of this debugger, but it is claimed that its response time is almost the same as regular debuggers. Obviously, this claim cannot be true in general due to the huge overhead of re-simulation.

2.3.8 JIVE [Czyz2007]

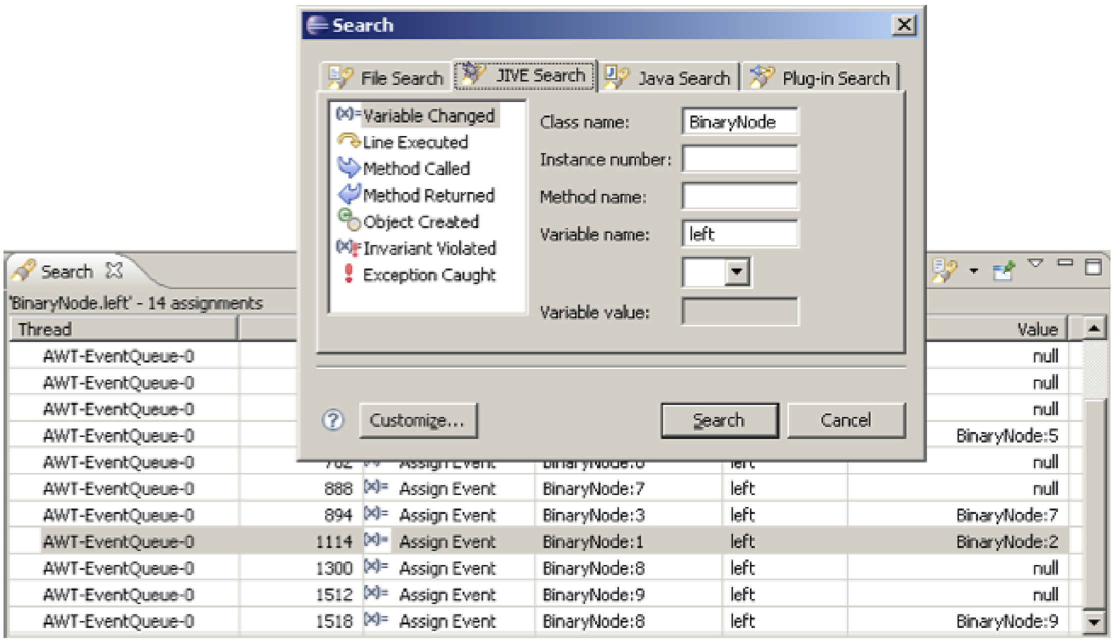
Jive is a capture-replay debugger that supports bidirectional queries on an execution. In terms of expressiveness, JIVE queries are slightly more expressive than breakpoints (Figure 2.3), but they are processed offline. They allow developers to search over the execution history for causes of a wrong value or an error. It also depicts the program state through an enhanced object diagram, and the history of execution as a sequence diagram.

2.3.9 TOD [Pothier2007]

A more scalable approach to capture-replay debugging has been proposed by Pothier et al. Their back-in-time debugger, *TOD*, addresses the space problem by storing execution events in a distributed database. In addition to bidirectional stepping, TOD also provides a *why?* link next to each field, which allows developers to navigate to the program state where the field value was set (Figure 2.4). TOD incurs slightly less overhead than ODB, 22 to 176 more times than the original execution.

2.3.10 Compass [Lienhard2008, Lienhard2009]

Compass is a capture-replay debugger, which is built on top of a modified virtual machine that tracks the flow of objects. Compass attempts to address the scalability issues of ODB by keeping only parts of the history that are needed for answering question about live objects. It replaces every object reference with an alias object. Therefore an object flow history is kept in the heap memory and it is removed by the garbage collector when there is no active reference


Search

File Search | JIVE Search | Java Search | Plug-in Search

- Variable Changed
- Line Executed
- Method Called
- Method Returned
- Object Created
- Invariant Violated
- Exception Caught

Class name:	BinaryNode
Instance number:	
Method name:	
Variable name:	left
Variable value:	

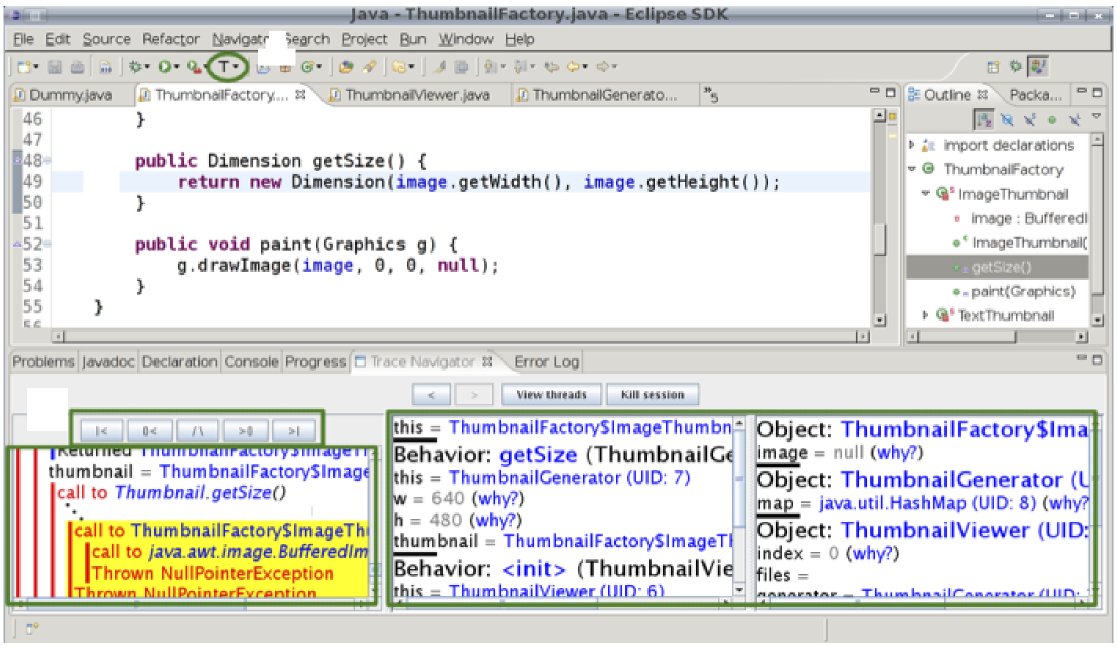
Customize... | Search | Cancel

Search

'BinaryNode.left' - 14 assignments

Thread					Value
AWT-EventQueue-0					null
AWT-EventQueue-0					null
AWT-EventQueue-0					null
AWT-EventQueue-0					BinaryNode:5
AWT-EventQueue-0	[illegible]	Assign Event	[illegible]	left	null
AWT-EventQueue-0	888	Assign Event	BinaryNode:7	left	null
AWT-EventQueue-0	894	Assign Event	BinaryNode:3	left	BinaryNode:7
AWT-EventQueue-0	1114	Assign Event	BinaryNode:1	left	BinaryNode:2
AWT-EventQueue-0	1300	Assign Event	BinaryNode:8	left	null
AWT-EventQueue-0	1512	Assign Event	BinaryNode:9	left	null
AWT-EventQueue-0	1518	Assign Event	BinaryNode:8	left	BinaryNode:9

Figure 2.3: JIVE query form and the result table.


Java - ThumbnailFactory.java - Eclipse SDK

```
        }

        public Dimension getSize() {
            return new Dimension(image.getWidth(), image.getHeight());
        }

        public void paint(Graphics g) {
            g.drawImage(image, 0, 0, null);
        }
    }
```

Returned ThumbnailFactory$ImageT
thumbnail = ThumbnailFactory$Image
call to *Thumbnail.getSize()*
call to ThumbnailFactory$ImageTh
call to *java.awt.image.BufferedIm*
Thrown NullPointerException
Thrown NullPointerException

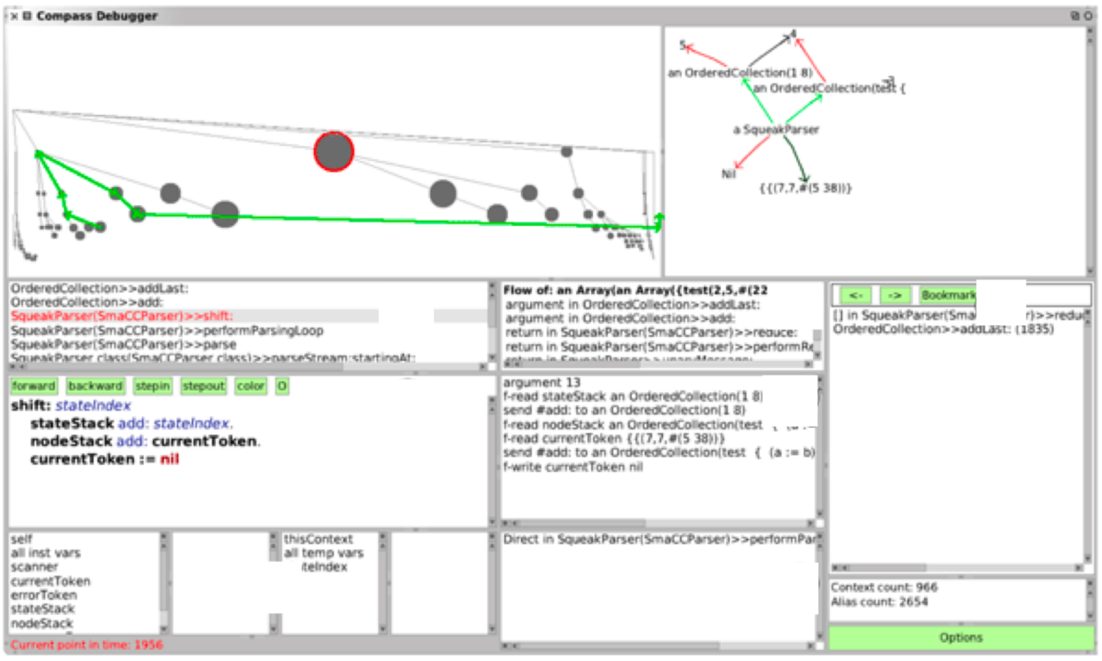
this = ThumbnailFactory$ImageThumbn
Behavior: getSize (ThumbnailGe
this = ThumbnailGenerator (UID: 7)
w = 640 (why?)
h = 480 (why?)
thumbnail = ThumbnailFactory$ImageT
Behavior: <init> (ThumbnailVie
this = ThumbnailViewer (UID: 6)

Object: ThumbnailFactory$Ima
image = null (why?)
Object: ThumbnailGenerator (U
map = java.util.HashMap (UID: 8) (why?
Object: ThumbnailViewer (UID:
index = 0 (why?)
files =
generator = ThumbnailGenerator (UID:

Figure 2.4: The screenshot shows TOD and its *why?* feature.

to the object anymore(Figure 2.5).

We name at least three main issues that prevent practitioners from employing this approach. First, this approach requires significant virtual machine modification, which seems impractical for most widely used programming languages such as Java and C#. Second, although this approach incurs less runtime overhead (7 times) in comparison to omniscient debuggers, it adds memory overhead. Third, Compass supports only certain backward navigations. For example, control flow dependencies are not stored and thus are missing.



```
OrderedCollection>>addLast:
OrderedCollection>>add:
SqueakParser(SmaCCParser)>>shift:
SqueakParser(SmaCCParser)>>performParsingLoop
SqueakParser(SmaCCParser)>>parse
SqueakParser class(SmaCCParser class)>>parseStream:startingAt:
```

```
Flow of: an Array(an Array({test(2,5,#(22
 argument in OrderedCollection>>addLast:
 argument in OrderedCollection>>add:
 return in SqueakParser(SmaCCParser)>>reduce:
 return in SqueakParser(SmaCCParser)>>performRe
```

```
[] in SqueakParser(Sma          r)>>redu
OrderedCollection>>addLast: (1835)
```

```
shift: stateIndex
    stateStack add: stateIndex.
    nodeStack add: currentToken.
    currentToken := nil
```

```
argument 13
f-read stateStack an OrderedCollection(1 8)
send #add: to an OrderedCollection(1 8)
f-read nodeStack an OrderedCollection(test
f-read currentToken {{(7,7,#(5 38))}
send #add: to an OrderedCollection(test  {  (a := b)
f-write currentToken nil
```

```
self
all inst vars
scanner
currentToken
errorToken
stateStack
nodeStack
```

```
thisContext
all temp vars
 teIndex
```

```
Direct in SqueakParser(SmaCCParser)>>performPar
```

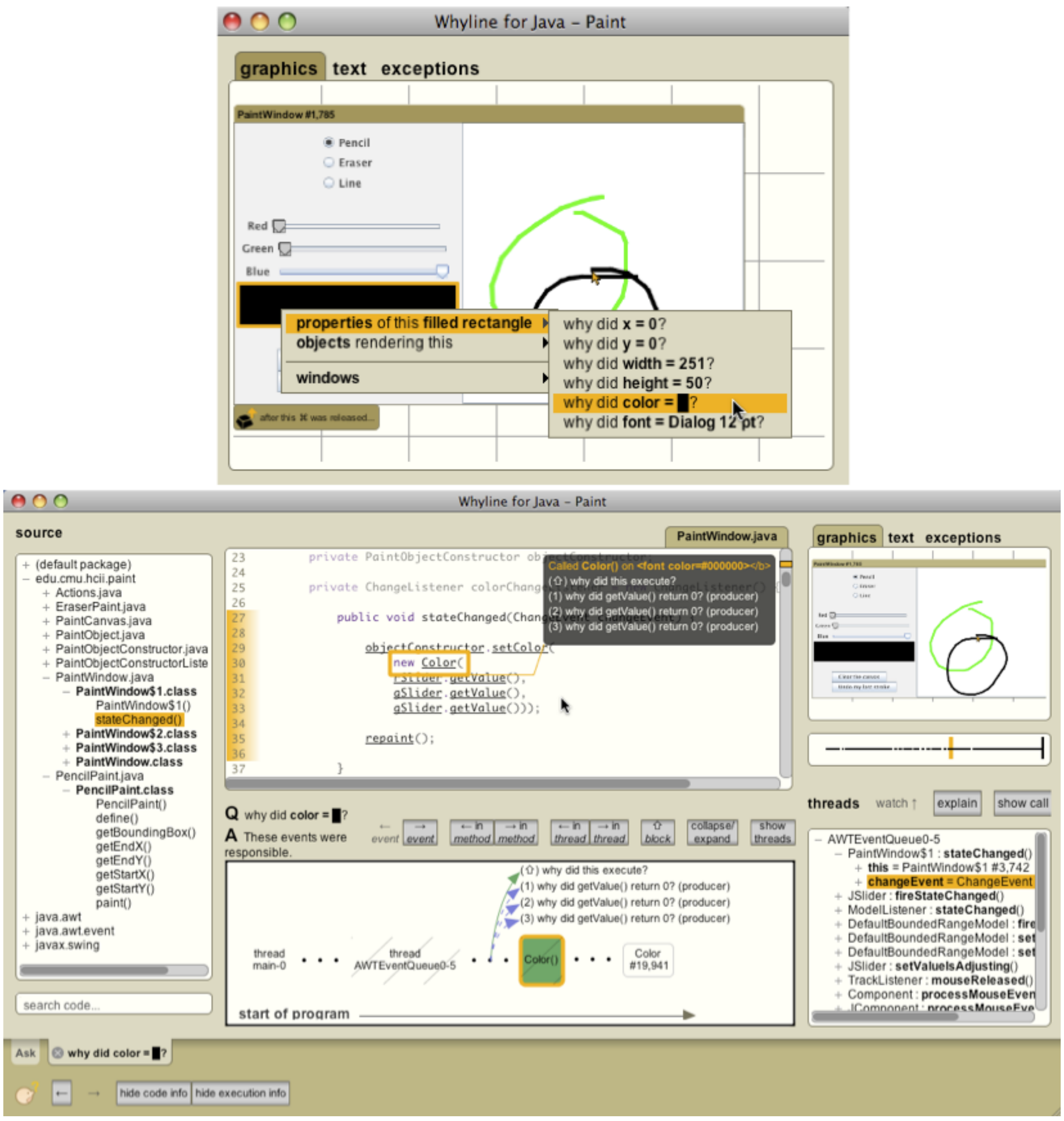
```
Context count: 966
Alias count: 2654
```

```
Current point in time: 1956
```

Figure 2.5: A Compass Screenshot demonstrating its flow-centric navigation feature.

2.3.11 Whyline [Ko2008]

Whyline is again a capture-replay debugger that keeps the execution history in memory. However, it provides richer queries than most back-in-time debuggers : it lets a user select questions about why some behavior did or did not occur. These questions are automatically generated based on a combination of static and dynamic analysis, and can deal not only with the internal state of the program (i.e., memory locations, control flow), but also with its textual and graphical output, down to individual pixels (Figure 2.6). Whyline's tracing overhead, like other capture-replay tools, is dependent on the nature of program and its event-per-second execution event ratio. The tracing time for executions with more than 10 to 35 million events is from 5 to 15 times more than the original execution time.



```
    private PaintObjectConstructor objectConstructor;

    private ChangeListener colorChangeListener = new ChangeListener() {

        public void stateChanged(ChangeEvent changeEvent) {

            objectConstructor.setColor(
                new Color(
                    rSlider.getValue(),
                    gSlider.getValue(),
                    gSlider.getValue()));

            repaint();

        }
```

Called Color() on <font color=#000000></b>
(⇧) why did this execute?
(1) why did getValue() return 0? (producer)
(2) why did getValue() return 0? (producer)
(3) why did getValue() return 0? (producer)

Q why did color = █?
A These events were responsible.

```
(default package)
edu.cmu.hcii.paint
  Actions.java
  EraserPaint.java
  PaintCanvas.java
  PaintObject.java
  PaintObjectConstructor.java
  PaintObjectConstructorListe
  PaintWindow.java
    PaintWindow$1.class
      PaintWindow$1()
      stateChanged()
    PaintWindow$2.class
    PaintWindow$3.class
    PaintWindow.class
  PencilPaint.java
    PencilPaint.class
      PencilPaint()
      define()
      getBoundingBox()
      getEndX()
      getEndY()
      getStartX()
      getStartY()
      paint()
java.awt
java.awt.event
javax.swing
```

```
AWTEventQueue0-5
  PaintWindow$1 : stateChanged()
    this = PaintWindow$1 #3,742
    changeEvent = ChangeEvent
  JSlider : fireStateChanged()
  ModelListener : stateChanged()
  DefaultBoundedRangeModel : fire
  DefaultBoundedRangeModel : set
  DefaultBoundedRangeModel : set
  JSlider : setValueIsAdjusting()
  TrackListener : mouseReleased()
  Component : processMouseEven
  JComponent : processMouseEve
```

Figure 2.6: A whyline screenshot demonstrates how it works. On the top, a question about the color of a rectangle using the Whyline for Java. On the bottom, the corresponding answer, showing the source of the black color and the causes of its creation.

2.3.12 Blink [Lee2009]

Blink is a debugger for mixed-language programs. The standard debugger features are implemented in Blink by composing single-environment debuggers through an intermediate agent. The mixed-environment debugger consists of a controller and one driver for each single-environment component debugger. The execution overhead of this debugger can reach 21 percent of the original execution time.

2.3.13 Causeway [Stanley2009]

Causeway is a capture-replay debugger for distributed programs. It follows the flow of messages across process and machine boundaries in order to support integrated navigation across multiple orders (Figure 2.7). *FireDetective* [Matthijssen2010] is another tool, with a similar technique, which is built for debugging Ajax applications.

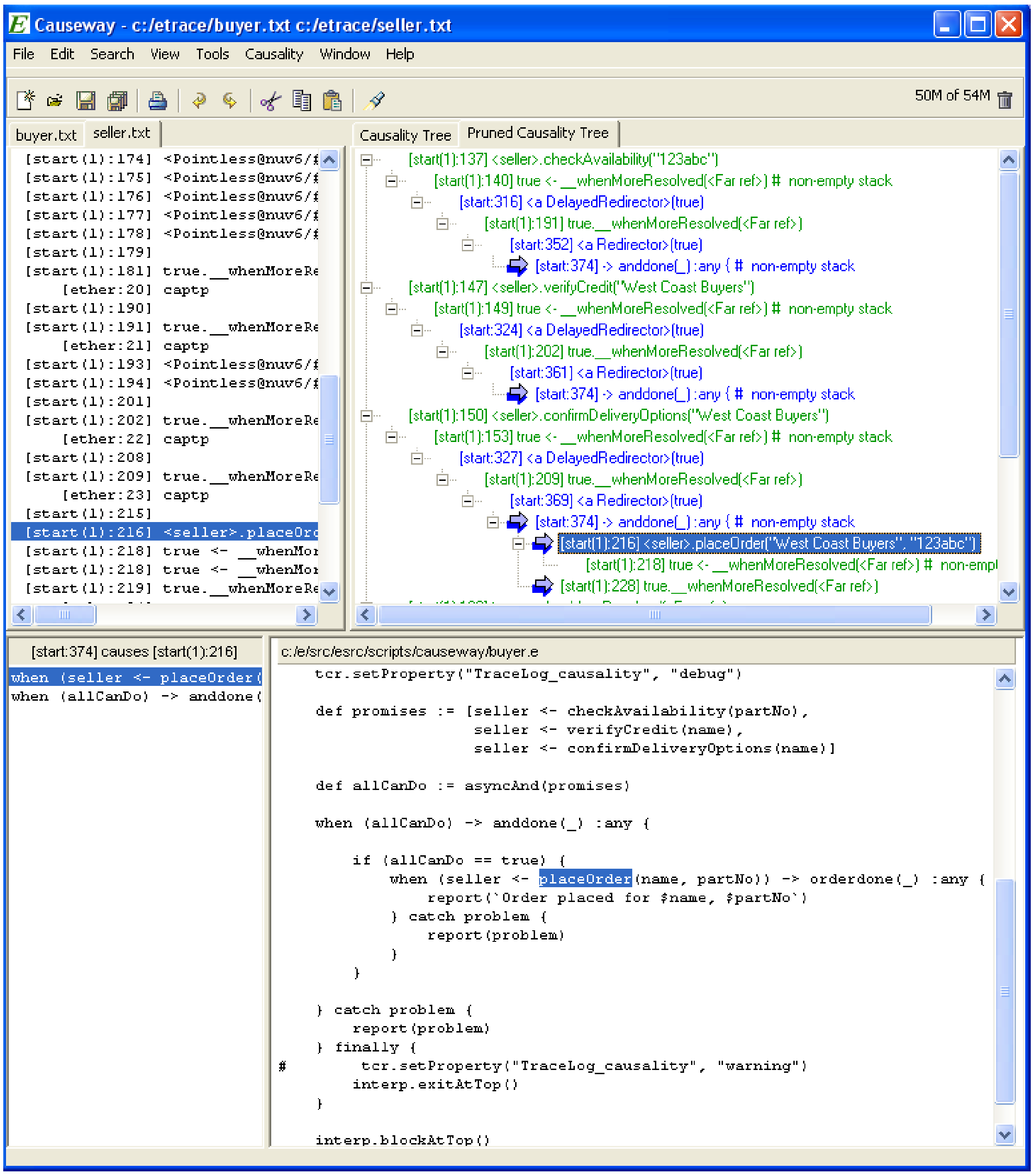
2.3.14 Graphical Breakpionts [Barton2010]

Graphical breakpoints are developed for web page debugging. By setting a graphical breakpoint on an element attribute or event, developers can inspect the program state when the element attribute is changed or the event is fired (Figure 2.8). This type of breakpoint provides a new dimension for inspection of the buggy execution.

Besides graphical breakpoints, other new types of breakpoints have also been invented. For example, control-flow breakpoints [Chern2007] allow developers to set breakpoints in a way very similar to the way they define joint points in aspect-oriented programming.

2.3.15 STIQ [Pothier2011]

To reduce the execution overhead in TOD, *STIQ* employs a hybrid method that consists of capture-replay and deterministic replay approaches. The STIQ process consists of four phases: trace capture, initial replay, summarized indexing, and querying. In the first phase, it records all outcomes of the non-deterministic operations. Then, an initial replay is performed to obtain a partial trace for bounded-size execution blocks. In the third phase, it indexes the stored information of each execution block. Finally, to answer a query, it first locates the corresponding execution block, then deterministically replays the block to determine the exact answer. Compared to TOD, STIQ significantly reduces the index size, but causes 10 to 30 times more execution overhead for trace capturing, which is still too much to be used in practice. Moreover, STIQ needs initial replay after the capturing phase, and an execution replay to answer each query. It is also inherited several problems that exist in deterministic replay approaches.


Causeway - c:/etrace/buyer.txt c:/etrace/seller.txt
File Edit Search View Tools Causality Window Help
50M of 54M
buyer.txt
seller.txt
Causality Tree
Pruned Causality Tree
[start:374] causes [start(1):216]
c:/e/src/esrc/scripts/causeway/buyer.e

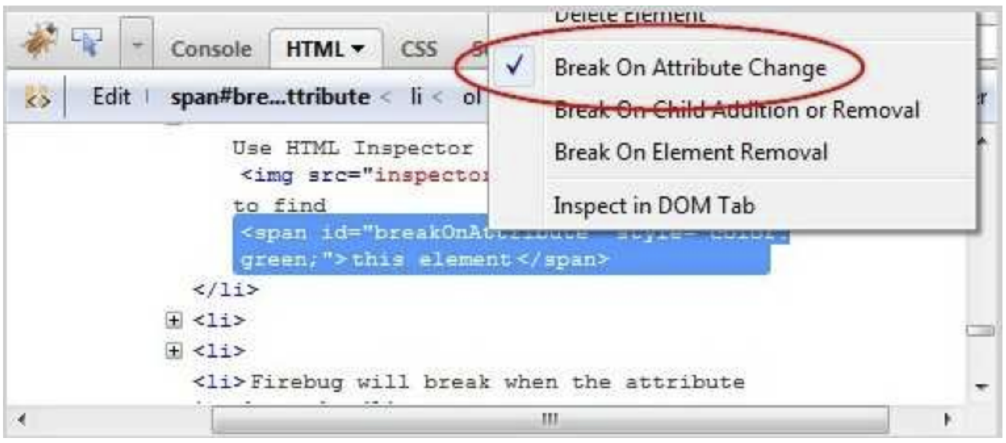
```
[start(1):174] <Pointless@nuv6/f
[start(1):175] <Pointless@nuv6/f
[start(1):176] <Pointless@nuv6/f
[start(1):177] <Pointless@nuv6/f
[start(1):178] <Pointless@nuv6/f
[start(1):179]
[start(1):181] true.__whenMoreRe
     [ether:20] captp
[start(1):190]
[start(1):191] true.__whenMoreRe
     [ether:21] captp
[start(1):193] <Pointless@nuv6/f
[start(1):194] <Pointless@nuv6/f
[start(1):201]
[start(1):202] true.__whenMoreRe
     [ether:22] captp
[start(1):208]
[start(1):209] true.__whenMoreRe
     [ether:23] captp
[start(1):215]
[start(1):216] <seller>.placeOrd
[start(1):218] true <- __whenMor
[start(1):218] true <- __whenMor
[start(1):219] true.__whenMoreRe
```

```
[start(1):137] <seller>.checkAvailability("123abc")
    [start(1):140] true <- __whenMoreResolved(<Far ref>) #  non-empty stack
        [start:316] <a DelayedRedirector>(true)
            [start(1):191] true.__whenMoreResolved(<Far ref>)
                [start:352] <a Redirector>(true)
                    [start:374] -> anddone(_) :any { #  non-empty stack
[start(1):147] <seller>.verifyCredit("West Coast Buyers")
    [start(1):149] true <- __whenMoreResolved(<Far ref>) #  non-empty stack
        [start:324] <a DelayedRedirector>(true)
            [start(1):202] true.__whenMoreResolved(<Far ref>)
                [start:361] <a Redirector>(true)
                    [start:374] -> anddone(_) :any { #  non-empty stack
[start(1):150] <seller>.confirmDeliveryOptions("West Coast Buyers")
    [start(1):153] true <- __whenMoreResolved(<Far ref>) #  non-empty stack
        [start:327] <a DelayedRedirector>(true)
            [start(1):209] true.__whenMoreResolved(<Far ref>)
                [start:369] <a Redirector>(true)
                    [start:374] -> anddone(_) :any { #  non-empty stack
                        [start(1):216] <seller>.placeOrder("West Coast Buyers", "123abc")
                                [start(1):218] true <- __whenMoreResolved(<Far ref>) #  non-empt
                            [start(1):228] true.__whenMoreResolved(<Far ref>)
```

```
when (seller <- placeOrder(
when (allCanDo) -> anddone(
```

```
    tcr.setProperty("TraceLog_causality", "debug")

    def promises := [seller <- checkAvailability(partNo),
                     seller <- verifyCredit(name),
                     seller <- confirmDeliveryOptions(name)]

    def allCanDo := asyncAnd(promises)

    when (allCanDo) -> anddone(_) :any {

        if (allCanDo == true) {
            when (seller <- placeOrder(name, partNo)) -> orderdone(_) :any {
                report(`Order placed for $name, $partNo`)
            } catch problem {
                report(problem)
            }
        }

    } catch problem {
        report(problem)
    } finally {
#        tcr.setProperty("TraceLog_causality", "warning")
        interp.exitAtTop()
    }

    interp.blockAtTop()
```

Figure 2.7: A Causeway screenshot demonstrates its user interface. In the upper left pane, each tab represents the full order of events recorded by each process. This gives a “follow the process” view, common to conventional distributed debuggers. In the upper right pane, there is an alternative “follow the conversation” outline view, in which each event expands to show the events it causes.


Console
HTML
CSS
Edit
span#bre...ttribute < li < ol
Delete Element
Break On Attribute Change
Break On Child Addition or Removal
Break On Element Removal
Inspect in DOM Tab

```
    Use HTML Inspector
     <img src="inspecto
    to find
    <span id="breakOnAt
    green;">this element</span>
  </li>
  <li>
  <li>
  <li>Firebug will break when the attribute
```

Figure 2.8: A Firebug screenshot demonstrates the graphical breakpoints feature.

2.3.16 Other Tools

Unstuck [Hofer2006], similar to ODB, stores the execution trace in RAM, in the same process as the debugged program. Amber/Chronicle [OCallahan2006] is a back-in-time debugger for native Linux programs, designed to deal with large execution traces. This tool is used to debug the Firefox web browser. The runtime overhead of trace capture for Amber is 300 more times than the original execution.

2.4 What Can Be Improved?

An improvement of widely-used debugging approaches must have one of the following effects: *reduce the debugging time, increase the quality of diagnosis* or *facilitate collaboration*[6]. Based on this description, we identify ten potential directions for improving the debugging practice. We describe each direction either as a solution to a known problem or as support for a feature that facilitates the developer's job.

Manual Re-Execution

Both traditional debugging approaches are based on re-execution, i.e., the buggy execution must be reproduced again if more data are needed. As we discuss in the previous chapter, debugging can be done during development, testing or maintenance. In the last two cases, a test-case is usually available for automating re-execution. However, in the case of debugging during development, a complete test case does not usually exist. A developer has to either manually repeat the initialization and the data entry for every re-execution or create a new test-case.

[6]This parameter can actually be expressed in terms of the first and second parameters for other developers, or for the same developer in another debugging session.

Problem or Need	Description	Reduces Debugging Time	Increases Diagnosis Quality	Facilitates Collaboration
Manual Re-Execution	In cases where a test case is not available, manual re-Execution is not efficient.	-		
Prolonged Execution	The re-execution time directly contributes to the debugging time. A prolonged execution can significantly increase the debugging time.	-		
Lost Debugging State	After a re-execution, a developer loses the debugging state. For example, they have to re-locate the log record they were inspecting before re-execution.	-		
Guesswork	A developer has to speculate the location of breakpoints and log statments.	-		
Irrelevant Data	A developer has to deal with and analyze irrelevant data before obtaining the relevant data.	-	-	
Overloaded Developer Memory	A developer has to memorize and keep important pieces of information in mind.	-	-	
Backward Navigation	A developer usually needs to inspect the history of program states.	+		
Multi-Environment Support	Widely-used approaches are very limited for debugging applications that are using more than one programming language or running on several layers. They are also not effective for debugging distributed programs.	+		
Diagnosis Integrity Check	A developer should be able to check the correctness of diagnosis and should receive early feedback if the diagnois is no longer valid.		+	
Debugging Artifacts	A developer should have artifacts for debugging, which let them document, re-use and collaborate the debugging task.			+

Table 2.1: The table lists problems/needs in the widely-used debugging approaches and their relation to three productivity parameters.

Prolonged Execution

A developer has to re-execute a buggy program several times before locating the defects. The re-execution time directly contributes to the debugging time. A prolonged execution can significantly increase the debugging time.

Lost Debugging State

A developer loses the state of debugging on the buggy execution whenever the program is re-executed and the developer has to regain it on the new instance of the buggy execution. For example, suppose that a developer, after a long inspection of log files, identifies a specific log record, but the collected data are not sufficient. Therefore, the developer updates the log statements and re-executes the program. A new set of log files are created. To continue the inspection, the developer first has to locate the same log record in the new log files.

Guesswork

A developer has to speculate the location of the log statements, the data to be collected, and the location of breakpoints. They must employ their expertise, search through the source files and be hopeful to finally insert a useful log statement or set an appropriate breakpoint after a few trials.

Irrelevant Data

After inserting log statements or setting breakpoints, developers still have to analyze huge trace files or go through tedious repeated cycles of pause, check, and resume. In other words, developers have to deal with much unnecessary data before they can obtain the desired piece of information.

Overloaded Developer Memory

The traditional debugging approaches rely excessively on the developer's memory. A developer has to keep in their minds the state of debugging, including the results of the previous inspections, the assumptions they made in the past inspection, the overall progress, and their current strategy.

Backward Navigation

As a common strategy for locating defects, a developer starts from the bug symptoms and traces the execution backwards, moving from a point in the program execution where a value appears to be incorrect back to the point where the new value was set. Unfortunately, backward navigation on the buggy execution is complicated[7] or in some cases not possible in the traditional approaches.

Multi-Environment, Multi-Language, Distributed Programs Support

Traditional approaches are very limited for debugging applications that run in different environments, on several layers or that use more than one programming language. They are also not effective for debugging distributed programs.

Diagnosis Integrity Check

Developers must be informed if their assumptions or findings—due to non-determinism or changes in the execution environment, configuration files, or source files—are not valid anymore in the new execution. Unfortunately, none of the two traditional approaches are able to inform the developer about the inconsistencies. So, a developer simply continues the inspection and probably obtains wrong results.

Debugging Artifacts

Developers need debugging artifacts in order to document and recover the debugging state, for example to resume debugging in another session, or to debug a reported bug that has already been fixed. Unfortunately, the whole process of locating defects does not produce any artifacts except a set of log statements or breakpoints, which are usually removed at the end of a debugging session.

2.4.1 The Response of New Approaches

Each new approach or tool we have discussed in 2.2 and 2.3 attemps to respond to one or a number of the above needs. “Instruction/Event indexing” and “capture-replay” try to mitigate the re-execution issues and provide backward-navigation. “Query-based” and “program slicing” reduce guesswork and the amount of irrelevant data. “Algorithmic debugging” provides a systematic approach to debugging (table 2.2).

We name *scalability* as the main challenge for these approaches; they are not effective or not efficient in the case of large traces. The other challenge is that these methods are not as *flexible* as widely-used approaches and they can not be easily integrated with them.

[7]A developer can inspect past program states by looking at the trace in the log-based approach, or by rewinding from an earlier instruction in the call stack in the breakpoint-based approach.

	Manual Re-execution	Prolonged Execution	Lost Debugging State	Guesswork	Irrelevant Data	Overloaded Developer Memory	Backward Navigation	Multi-environment Support	Diagnosis Integrity Check	Debugging Artifact	
Approaches	**Target Problems**										**Challenges**
Instruction/Event Indexing	•	•	•				•				execution slow down, identical re-execution
Capture-Replay	•	•	•				•				significant execution overhead, missing aspects of execution in the trace, efficient navigation on the trace
Query-based				•	•						providing convenient ways of defining/using queries, query response time, integration to other approaches
Program Slicing				•	•						efficient algorithms for computing slices, reducing the slice size
Algorithmic Debugging						•			•	•	needs the execution trace, reducing the number of questions, hard-to-answer questions

Table 2.2: New debugging approaches, problems they tackle and their challenges/limitations.

2.5 Summary

In this chapter, we discussed and analyzed widely-used debugging approaches and identified 10 important directions for improving these approaches. We described key ideas employed for tackling these problems. We also provided a brief summary of new debugging tools, their features and limitations.

What saves a man is to take a step. Then another step.

— C. S. Lewis

3 The Fundamentals of Locating Defects

Locating the defects that cause a bug, along with providing the explanation of how these defects eventually cause the bug symptoms, are at the center of debugging process. It is no surprise that this stage is often reported by developers as the most laborious part of debugging [Eisenstadt1997]. The root causes of this difficulty, however, remain concealed and are not well understood. A profound analysis of the process of locating defects can reveal the answer to why this stage of debugging is so hard and time consuming, and can probably lead us to improvements in the employed tools and techniques.

The common approach to locating defects is to inspect the buggy execution. Developers compare at different execution points the program state with the initial expectations to understand where and why they begin to diverge. The developer's knowledge, strategy, and decisions are the main elements in forming the process structure. Hence, a thorough analysis of the process of locating defects cannot be achieved without understanding how developers approach the problem, how their mental model is shaped during the process, and how this mental model influences their decisions.

In this chapter, we first define the basic terms and concepts frequently used in this dissertation. We then describe the problem of locating defects based on these definitions. In the rest of the chapter, we study this problem and provide answers to the following questions about the process of locating defects:

1. What is the developer's mental model and how is it shaped during the process?
2. What are the developer's basic actions, and how do they form the structure of process?
3. What is the developer's basic strategy, and how does it drive the sequence of developer's actions?
4. What are the common approaches to this strategy, and what are their problems?
5. Based on the answers to the above questions, how can debuggers be improved to better support developers?

3.1 Basic Definitions

The terms defined in this section are restricted to those that have to be defined in a new form, such as *defect* and *fault*, or for the first time such as *expected execution* and *correct/buggy execution point*.

Target Debugging Program (Debuggee)

In real systems, several programs work together to perform a specific task. These programs can have varied responsibilities and work in different layers. For example, in a regular web application, many programs such as operating systems, application servers, web browsers, and database managers work together to serve user requests. When developers debug, they have to trust most of these programs (e.g., the operating system, web browser, and database manager) and focus only on one or a few ones (e.g., the client, the server, or both). The *target debugging program* or *debuggee* is the collection of programs or components (i.e., parts of programs) that are not trusted and are being investigated by the developer. The debuggee might change in a debugging session as the developer readjusts the suspect programs.

Execution Point

Developers understand an execution as a sequence of program states. Depending on the boundaries of the target debugging program, every state consists of a program *internal* state (i.e., the call stacks and memory heap states), program *external* state (e.g., the database state), input and output states. We can assume that the execution runs from one state to the next by applying an instruction to the current state. Note that we can deal with event-driven, multi-thread, multi-process, or distributed programs in a similar way by serializing the executed instructions. We call every state in the execution an *execution point* (Figure 3.1).

Bug Symptom

A *bug symptom* is an unexpected or suspicious event, state or behavior, which appears when the program is running. For example, in a user interface, a label with a wrong text or color is a bug symptom. A bug symptom can vary from serious issues, such as program unresponsiveness or lost data, which can totally disturb the program functionality, to unpleasant behaviors such as prolonged program execution or annoying error messages.

Buggy Execution, Reproducible

A *buggy execution* is an execution of a program that manifests one or more bug symptoms. A buggy execution is *reproducible* when there is a test case for the program that reproduces the same bug symptoms in every test case run. The test case usually performs three main tasks :

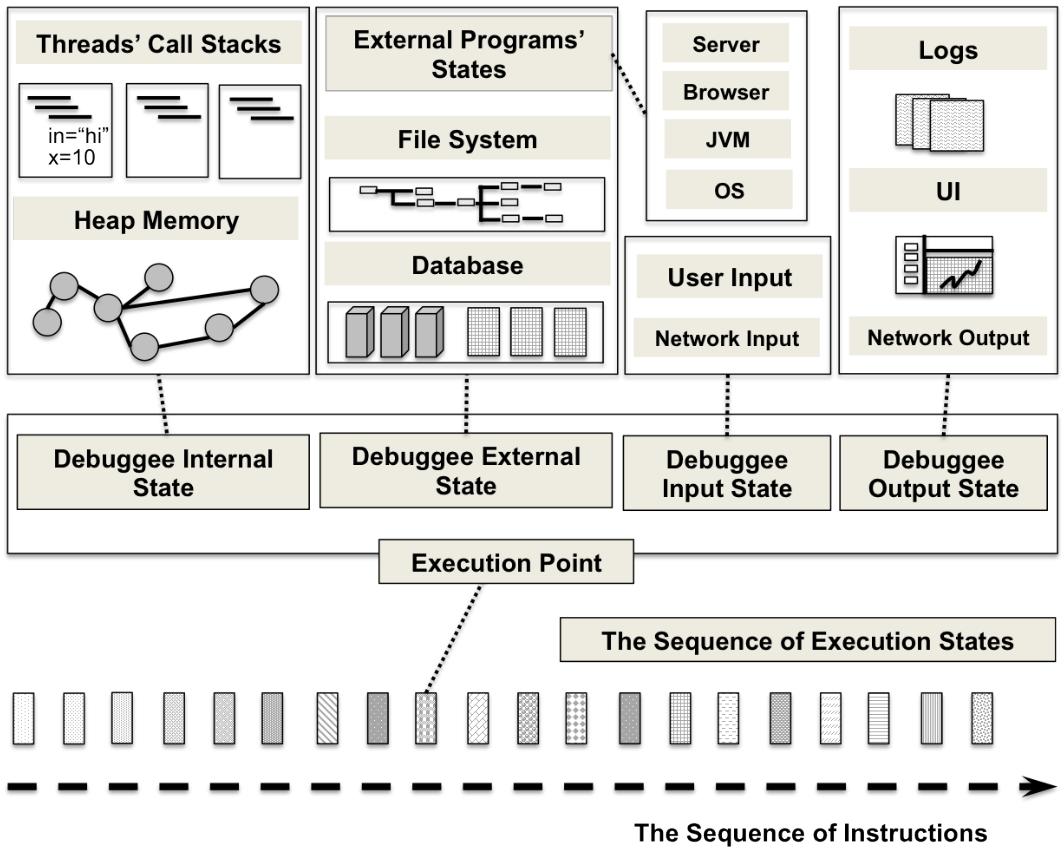


Figure 3.1: Developers understand execution as a sequence of states. Every execution state includes a program(s) internal, external, input and output states.

(1) restores the execution state to an initial state, (2) applies a set of inputs, actions and events to the debuggee when it is running, and (3) stops when the bug symptoms appear. We call a test case run a *reproduction cycle* of the buggy execution.

Note that the constraint of "the bug symptoms manifestation in *every* test case run", does not restrict the definition. In fact, having a test case, which occasionally manifests the bug symptoms, and the termination conditions for every test case run, we can always, by repeating the original test case execution, build a new test case that reproduces the bug symptoms in every run.

Of course, restoring exactly the same initial state in every reproduction cycle is almost impossible. In practice, developers often choose the parts of execution state that need to be restored in order to reproduce the bug symptoms. There is usually a trade-off between the accuracy of the restored initial state and the time needed to run one reproduction cycle. For example, in an event-driven application, developers have the option of starting the test case by re-firing the last event, instead of executing the application from the beginning. When this inner cycle reproduces the bug without confusing side effects, it provides shorter reproduction cycles.

Trace

A *full trace* of an execution is the stored data of all execution points. A *partial trace*, or simply a *trace*, differs from a full trace in that it does not contain all execution points or the whole state of each execution point. The data stored at one execution point is called a *tracepoint*.

Defect

A *defect* is the description provided by the developer that explains what causes a buggy execution to deviate from the expectations. As we discuss in the next section, a defect might not be directly mapped to any artifact. Therefore, a defect is usually described based on high-level concepts that are understandable to the program's developers. For example, a developer can describe a defect like this: "the bug happens because in the layout design, we did not anticipate very long pages." or "the bug happens because two users try to buy one item at the same time, which is not predicted in the system."

A defect can be in the program, the expectations or both (Figure 3.2); a defect can be due to wrong assumptions (e.g., the input has a specific format, the external library call releases all resources when it returns, etc.), unforeseen cases, design flaws or implementation mistakes (e.g., forgotten functions, typos, etc.).

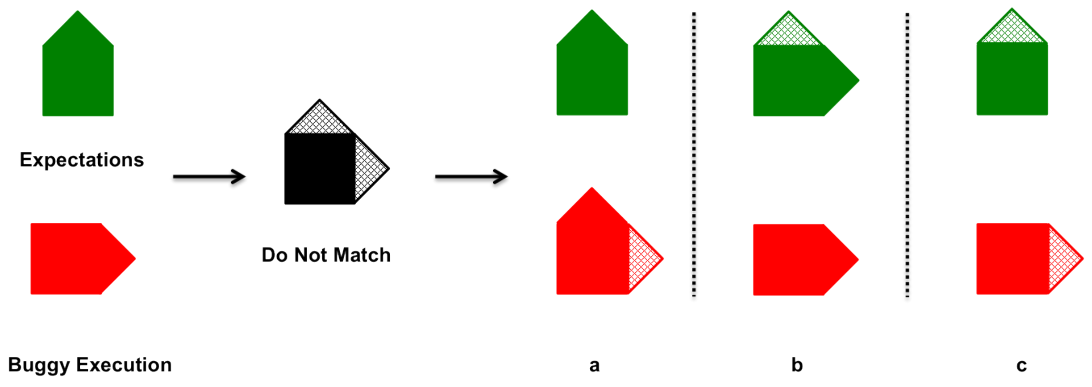


Figure 3.2: The expectations and what happens in the buggy execution do not match. There are issues in the program (case a), or in the developer's expectations (case b), or both (case c).

Expected Execution

An *expected* execution is a hypothetical execution in the developer's mind that is, according to the developer's knowledge, expected to happen instead of the buggy execution. It is very common that details of expected execution change in a debugging session due to the new knowledge acquired by the developer.

Correct Execution Point

We call an execution point a *correct* or *as-expected* execution point if the developer believes that the point is on the expected execution. In other words, the developer checks the state and execution point's location against a set of *sufficient* conditions. We will discuss how these conditions are defined by the developer in 3.3.

Buggy Execution Point

An execution point that is not on the expected execution is called a *buggy* execution point. In other words, the developer checks the state and execution point's location against a set of *necessary* conditions, i.e., the execution point is buggy if any of the necessary conditions is violated. The part of an execution state in a buggy point that differs or is missed from the expected execution, is called a *buggy state*. For example, if the value of an alias must not be `null` whereas it is, the alias value is in the buggy state.

Determined/Undetermined Execution Point

A correct or buggy execution is called a *determined* execution point. In contrast, an execution point that is not yet determined by the developer is called an *undetermined* execution point. An execution point status (correctness or buggyness) can change due to changes in the expected execution.

Fault

A *fault* occurs due to a defect. It is an execution point in the buggy execution where the defect deviates the buggy execution from the expected execution (Figure 3.3). In other words, a fault is the first buggy execution point after a correct execution point. It is conceivable that more than one fault exists in the buggy execution, either because the faults affect the execution temporarily and then their effect disappears or the developer's checks are insufficient or there are mistakes. We will see an example of insufficient checks in 3.5.

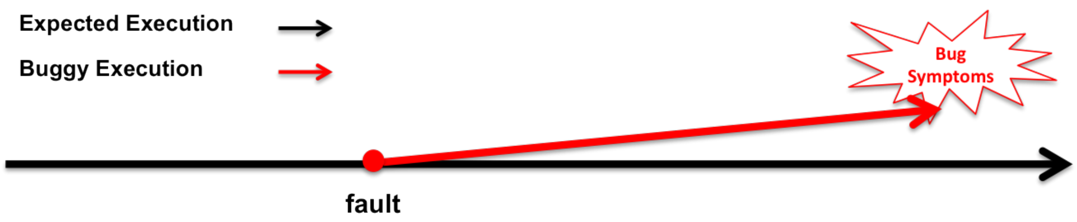


Figure 3.3: A fault is the execution point where the buggy execution deviates from the expected execution point.

Bug Transformer

We call any execution point after a fault a *bug transformer* if the buggy state changes from the previous execution point. For example, a bug transformer may change the buggy state from data flow to control flow (e.g., a condition check on a wrong value) or move a buggy value from memory to filesystem or database.

3.2 The Problem of Locating Defects

In order to analyze the process of locating defects, we need to clearly define the problem developers attempt to solve. This is the initial definition of a locating-defects problem: *Given a reproducible buggy execution, locate all defects in the program that must be eliminated in order to obtain the expected execution.* The solution to this problem consists of a set of defects and how these defects taint the program execution one after another before eventually causing the bug symptoms. We first assume that there is only one defect. Then we discuss the cases where more defects influence the program execution.

A simple local defect (e.g., a typo) can evidently be mapped to one or a few locations in the source code. However, there is not always a clear mapping from defects to source code. For example, in order to correct a design flaw, a large number of locations in the source code must be changed, whereas perhaps none of these locations individually represents the defect. Obviously when the defect exists due to omitted code, there is no source code that the defect could be mapped to.

Although defects may not be clearly mapped to source code, faults caused by defects can always be mapped to specific execution points. As mentioned in 3.1, a fault is the first buggy execution after a correct execution point. Therefore the last executed instruction before the fault is the instruction that can provide information about the defect to the developer.

Developers often understand a defect when they isolate a corresponding fault and obtain the transformers that connect the fault to the bug symptoms. When there is more than one defect, developers often find the first defect by locating a fault on the buggy execution. Then they eliminate the defect or assume the defect eliminated, which alters the expected execution, and locate a new fault in the new buggy execution or against the new expected execution. This process continues until all faults, therefore bug symptoms, are eliminated (Figure 3.4).

Based on the above discussion, a developer's search for defects can often be reduced to a search for faults and bug transformers in the buggy execution. Hence, we can reformulate the locating defects problem in this new form: *Given a reproducible buggy execution, locate a fault and the bug transformers on the buggy execution.* There are two delicate points in this definition. First, the location of these points must be provided on the buggy execution and not just, for example, in the source code. Second, for each of these points the answer must explain why the point is buggy, i.e., it must specify the buggy state, and why it is not expected.

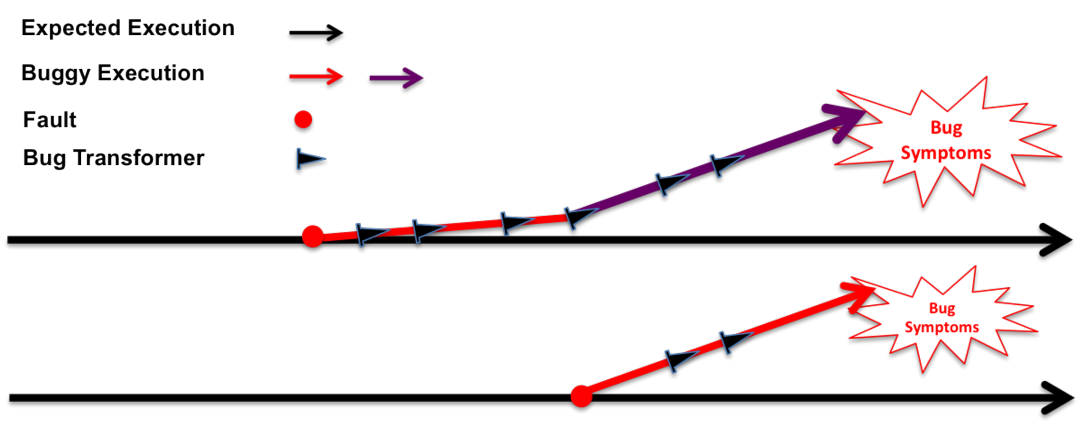


Figure 3.4: The upper drawing shows a case where two defects influence the buggy execution. After locating the first fault and eliminating the corresponding defect, the developer can look for the next fault (the lower drawing) to locate and eliminate the second defect.

In addition to the above definition, we need to also provide a relaxed version of the problem that is commonly used by developers: *Given a reproducible buggy execution, locate a fault on the buggy execution.* In this version, locating bug transformers is not necessary. Therefore, developers can save time by optimistically assuming that the located fault is the root cause of the bug symptoms. They try to directly eliminate the defect that causes the fault to hopefully obtain the expected execution afterwards. There is obviously a trade-off between the speed and the accuracy of diagnosis here.

3.3 The Developers' Mental Model

Locating defects is all about comparison; comparing what is expected to happen to what actually happens in the buggy execution. The expected part is understood from several sources: the developer's knowledge of the code and its intended behavior, the source code, the program execution, the requirements and the design documents, teammates, and the web. The actual execution, however, is mainly understood by inspection.

Locating defects is hard and time consuming because developers usually have to obtain both parts and compare them while they are changing. This explains why it is much easier for developers to debug their own code than the code that is written by other developers. As they know the design and rationale behind the source code, they do not have to search for the knowledge required to comprehend the expectations, and they can focus solely on inspecting the buggy execution.

During the process of locating defects, the developers's knowledge can be modeled in two sets of hypotheses and facts. The hypotheses set consists of two major categories: expectations, and defects. The expectations category is about the correct operation of the program. This category is large for the developer who wrote the code and it grows during debugging for developers who did not. The defects category contains ideas about what the defects are and how they manifest in the program state; and debugging is about keeping this category small and finally confirming it. The second set includes the facts learned by the developer from inspecting the buggy execution.

We illustrate this model by a simple example (Table 3.1). Suppose a buggy e-mail web client does not set the message's sender for a particular user (fact 1). This is the first fact that is known by the developer about the buggy execution. The developer also knows that once a user logs in, the server reads the user's data from the database and stores them in the session (hypothesis 1), and when the user sends a new message, the user's first and last name are used to set the message's sender (hypothesis 2).

In order to locate the defects, the developer starts inspecting the buggy execution. The developer first realizes that the sender property in the composed message object is `null` (fact 2). Through more inspections, the developer understands that the sender property is set by a value that is directly read from the database. In fact there is a column with name `COMPOSE_SENDER`, which contains this value (fact 3).

Fact 3 tells the developer that hypothesis 2 is incorrect and the sender value comes from the database, not the session (hypothesis 3). This fact implies that all execution points in the buggy execution are buggy and in fact the root fault is not in the buggy execution because the wrong sender value was in the database even before the buggy execution began (hypothesis 4). To confirm this hypothesis the developer checks the `COMPOSE_SENDER` value at the beginning of the reproduction cycle, and finds out that it is `null` (fact 4). After more investigations, the developer eventually understands that the `null` value was created when this user's data were imported from the old database to the new one.

The expected execution defined in Section 3.1 is equivalent to the set of hypotheses that describes the expectations. As the developer studies the buggy execution, new hypotheses are formed and execution points are selected based on these hypotheses. A correct execution point confirms some expectations about correct operation or it disproves some idea about how the defects manifest. A buggy execution point disproves some expectation about the correct operation or proves some hypothesis about how the defects manifest.

Of course, due to changes to the expected execution, the statuses of execution points might change (e.g., from correct to buggy or vice versa). A common case usually happens when developers are not able to obtain enough knowledge about a particular part of a program, or they are not confident about their comprehension, or they just want to continue with their guesses and save time. In such cases, they assume a set of hypotheses (e.g., a function call does not change a particular value or never returns a `null` value) and continue inspecting the buggy execution based on these hypotheses. It is usual that the developer later realizes that some of these hypotheses are not always true, or are incorrect, or do not lead to any useful results. Therefore, they have to replace them with a new set of hypotheses, and continue their search based on the new expectations.

Hypotheses (Expectations/Defects)	Facts (Observations)
H1- When a user logs in, the server reads user's data from the database and stores them in the session. H2- When the user sends a new message, the user's name and last name are used to set the message's sender.	F1- The sender field of messages sent from a particular user is empty!
H3- Hypothesis 2 is incorrect, the sender value comes from the database, not the session. H4-The root fault is not in the buggy execution since the wrong sender value was in the database before the buggy execution starts.	F2- When the composed message is sent the sender value is "null". F3-The sender field is set by a value that is directly read from the database. There is a special column with name COMPOSE_SENDER for this purpose. F4- The COMPOSE_SENDER value is null at the beginning of the reproduction cycle.

Table 3.1: This table shows the hypotheses and facts discovered by the developer in a debugging session. The left and right columns show the hypotheses, and the facts learned from buggy execution respectively.

3.4 The Structure of Process

If we record a developer's actions in a debugging session and then filter out all actions that are not about inspecting the buggy execution, what remains is a sequence of inspected execution points. In the e-mail web client example, introduced in the previous section, the developer inspects at least three execution points: P1, the execution point where the message is sent, P2, the execution point where the sender property is set, and P3, the execution point at the beginning of the reproduction cycle. For every execution point inspection, the developer must go through the following five steps:

1. Decide about the next execution point(s) to be inspected in order to form or verify/nullify hypotheses.
2. Select the execution point(s) on the buggy execution.
3. Obtain the program state at the execution point(s).
4. Select parts of the program state(s) for inspection.
5. Analyze or save the selected values.

The first step, deciding about the next execution point(s), happens in the mind of developer. A new execution point is selected by the developer in order to form new hypotheses, or to verify/nullify some of the assumed hypotheses. In the e-mail client case, the developer decides to inpect P1, in order to understand whether the problem occurs before or after sending the message. The outcome hypothesis is that the `null` value is set before sending the message. We discuss strategies used by developers to select the next execution point in the following sections.

After making the initial decision about the next execution point(s), the developer must locate the point(s) on the buggy execution. The developer has to describe the original concept in mind by the means that the debugging approach provides. For example, to locate P1 by breakpoints, the developer needs to find all functions in the source code that are used for sending messages, and to set a breakpoint on each of them. In a log-based approach, the developer must insert a log statement at the beginning of each of these functions. In both approaches the developer has to transform the original concept in mind to specific constructs (e.g., breakpoints or log statements). This task is not effortless. Moreover, the result of conversion is not usually accurate, i.e., the developer misses some of the execution points or gets unsought ones. Missing points lead to forming wrong hypotheses, and irrelevant points require the developer to iterate over the selected execution points and filter them out.

In the third step, the developer obtains the program state at the located execution point. In the fourth step, the developer selects parts of the program state that need to be inspected. For example, in breakpoint-based debugging, as the whole execution state is available at a breakpoint hit, no additional work is necessary for the third step. However, the developer has to manipulate the debugger via an interface to select the appropriate state for inspection. In log-based debugging, these two steps are reversed. The developer has to determine the data needed for inspection, inserts the log statement, and then reexecute the program. In the fifth step, the developer analyzes or saves values to form new hypotheses or verify/nullify the assumed hypotheses.

An important question raised here is, "Why does a developer inspect an execution point?" Based on the model provided for the developer's mental model, the developer inspects an execution point, either to understand how the program works (e.g., the heap/object structure, the execution scenario, etc.) and build up the expectations, or to check the state of the execution points against the expectations in order to locate the fault. In the remaining sections, we mainly focus on the second part, **fault localization**. Nonetheless, we briefly discuss the effect of changes in the expectaions on the fault localization process in Section 3.8.

3.5 Reducing the Suspicious Interval

We argue that in a closed interval that starts with a correct execution point and ends with a buggy execution point, there is always a fault. We prove this argument by contradiction. Assume that there is no fault in this interval. This implies that the execution points in this interval are either all correct or all buggy, which is simply not correct.

Developers implicitly employ the above argument in order to isolate a fault. They usually limit their search to an interval with these features—we call it *the suspicious interval*—and then try to locate new correct or buggy execution points in the interval in order to reduce the size of search space (Figure 3.5). In other words, if a developer identifies a correct execution point, then the fault must happen after that point, and if the developer identifies a buggy execution point, then the fault is either that point or happens before that point.

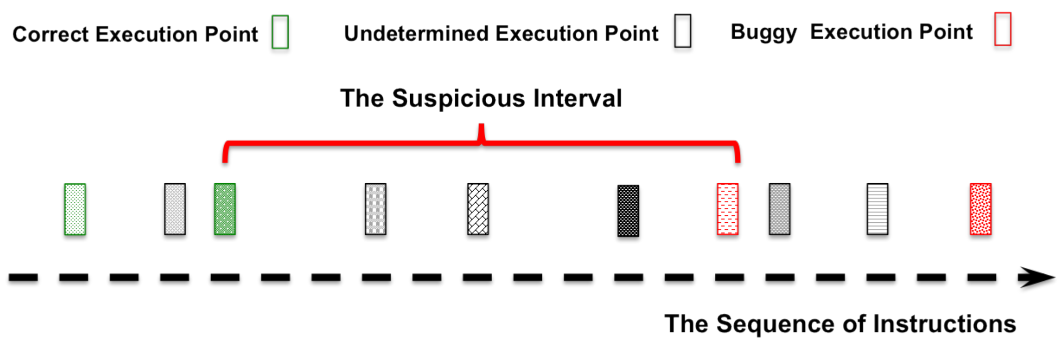


Figure 3.5: Developers limit their search to isolate the fault to the execution points in the suspicious interval, an interval that starts with a correct execution point and ends with a bugggy execution point.

The following example shows how developers choose and change the suspicious interval. Suppose a developer debugs a buggy program that stores data in a database. The bug symptom, an error, occurs because a `null` value is created and stored in the database and later, when the value is read from the database, it is converted to zero by the program, which is inconsistent with the program state and ultimately causes the error (Figure 3.6).

The developer starts with inspecting point **A** where the error appears. The initial suspicious interval starts at the beginning of the execution and ends at point **A**. The developer then selects point **B** to be inspected. As the wrong `null` value is in the database and the developer only checks the program internal state (insufficient check), this execution point looks correct. Therefore, the developer selects the interval from **B** to **A** as the second suspicious interval.

The developer reduces the length of this interval by locating more correct and buggy execution points in this interval. Finally, the developer identifies point **C** and understands that the wrong value is read from the database. As the database code is assumed bug-free, this implies that all execution points from the place where this value is stored to the database (point **D**) to point **C** are buggy. The developer locates point **D** and selects a new suspicious interval from the beginning of the execution to point **D**. The developer continues by reducing the size of this interval, until they finally locate point **E**, the true fault.

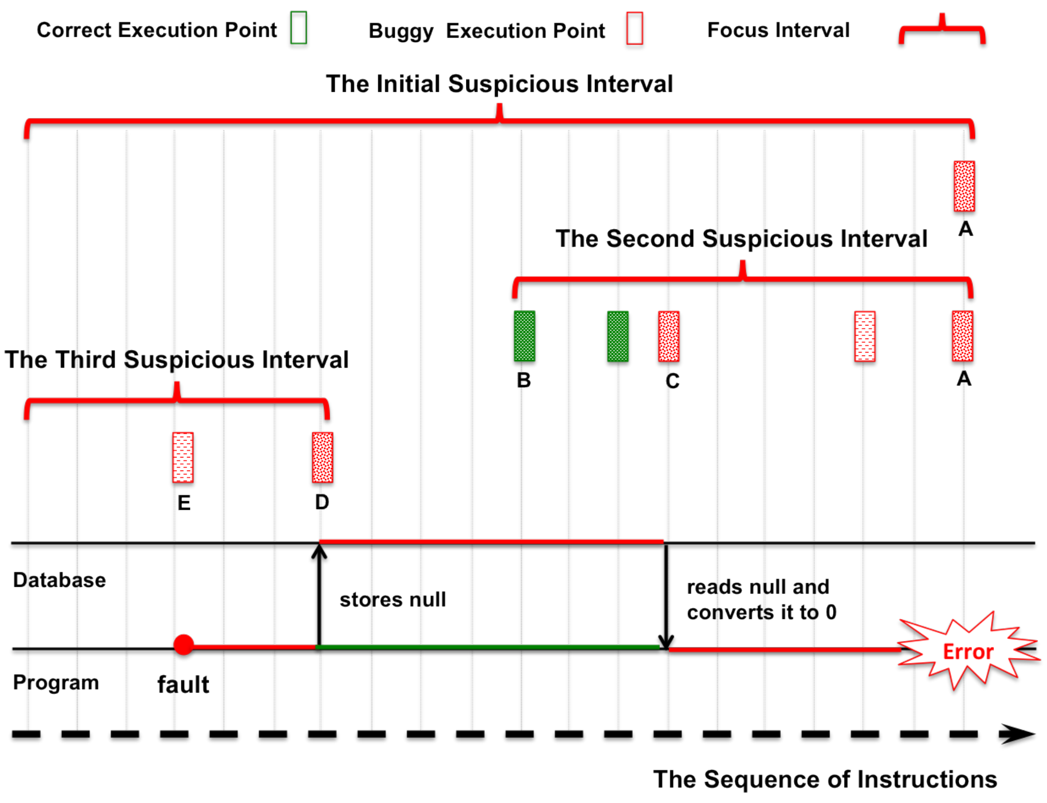

Correct Execution Point
Buggy Execution Point
Focus Interval
The Initial Suspicious Interval
A
The Second Suspicious Interval
B
C
A
The Third Suspicious Interval
E
D
Database
stores null
reads null and converts it to 0
Program
fault
Error
The Sequence of Instructions

Figure 3.6: Developers limit their search for isolating the fault to the execution points in the suspicious interval, an interval that starts with a correct execution point and ends with a buggy execution point.

3.6 The State Check Problem

In order to reduce the length of a suspicious interval, developers must determine whether the execution point is correct or buggy. This problem, which we call the *state check* problem, arises at almost every execution point inspection. Moreover, this is one of the main sources of difficulty in the process of locating defects. In fact, the problem of fault isolation could be reduced to a binary search if this problem could be easily solved for every execution point by the developer [Zeller2002].

To solve the check state problem, developers need the execution state and the location of the execution point in the execution scenario. The location is often very important because it lets the developer adjust the expectations (sufficient/necessary conditions) based on the execution point location. For example, the undo button in an editor application must be disabled when the editor window is initially opened, but this button must be enabled after the user performs any changes to the document. Therefore, the developer must know whether the document has just been changed to know the correct value for the button state.

One of the important factors in the approach used for selecting the subsequent execution point is the ability to solve the check state problem. For example, in cases where the program state is complex, developers can only check the state if they monitor state changes in short forward steps. This is particularly true when they deal with huge data collections (e.g., very long arrays) or complex data structures (e.g., red-black trees), which are difficult to analyze and check mentally.

In cases where developers are not able to clearly answer this problem (e.g., a suspicious execution point), they assume the execution point is either correct or buggy and continue with the right or the left interval. If they later observe that this assumption cannot be true or does not lead to any result they recommence and choose the other interval.

3.7 Subsequent Execution Point(s) Selection

We classify the approaches employed by developers to select the subsequent execution point and reduce the suspicious interval in three general categories: *controlled forwards navigation*, *causality backwards navigation*, and *search*.

3.7.1 Controlled Forward Navigation

A natural approach to reducing the length of a suspicious interval is to start from a correct execution point and to monitor the program's behavior and state changes until a problem is identified. The problem with this approach is that it requires a long time if the developer wants to check the program state at all execution points. To overcome this issue, developers monitor the execution progress with controlled steps until a buggy execution is found or the developer switches to another approach: The steps should not be so long that the developer cannot check and understand the execution state changes and should not be so short that it takes too much time.

At every step, the developer needs to know the program behavior during the step and the program state difference from the previous point; what is added, changed, and removed from the previous execution point. Of course, developers do not need all state changes to check the state. They often check a few particular values that are usually dependent on the step granularity. We call the data required to be collected in a step, *step summary*. Once developers identify a buggy execution in their forward navigation to locate the fault, they might employ causality backward navigations as we discuss in the next section, or they might try to renavigate the last step at a higher level of granularity with a shorter step (Figure 3.7).

In order to specify the next execution point, two main questions must be answered: What is the step? (or equivalently, what is the next point?), And what data must be collected during one step execution? Based on the developer's decision, steps with different granularities could be chosen. For example, to monitor an event loop execution, the developer could choose one loop iteration or the whole loop as one step. In the one loop iteration, collecting value changes

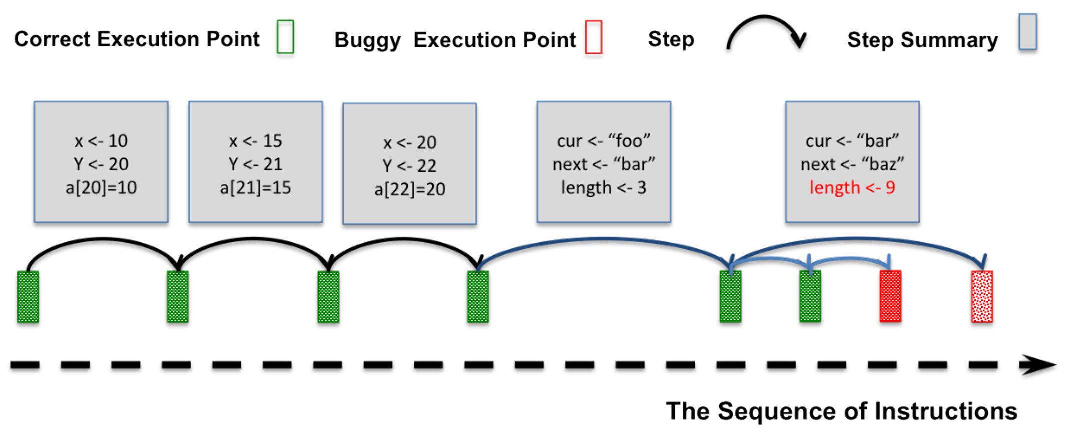


Figure 3.7: In controlled forwards navigation, developers determine the step and data to be collected during one step. Then they look for the step summary (the collected data) and the next execution point.

in every line might be necessary, whereas in the whole loop step, the developer might just need values of particular events.

Programs have some "waypoints", where one logical phase of work ends and another begins. One of the common choices for the next step is to the next waypoint. Depending on the level of granularity chosen by the developer, the next waypoint may vary. One reason that makes waypoints good choices for the subsequent point selection is that it is usually easier for developers to answer the check state problem at these points.

In breakpoint-based debugging, developers have to either use predefined steps (step-in, step-over, step-out) with very low granularity or simulate a step by breakpoints: They have to make sure that the next breakpoint hit, which does not provide the step summary, is not too far from the current point.

In log-based debugging, developers usually have many predefined log statements in different granularities. Hence, they can reuse steps that are already defined by these log statements. However, finding the right logging options that provide the appropriate granularity while do not generate too much data is often difficult. Moreover, they have still to insert new log statements to obtain required data that is not collected by the predefined log statements.

3.7.2 Causality Backward Navigation

Another natural approach to selecting the subsequent execution point is to start from a buggy point and locate the origins of the buggy state. Two different cases appear here; first, there are values that are wrong because the developer does not expect them (e.g., `null`, empty string, etc.), or expects a different value. Second, there are values that are totally missing (there is not even a wrong value).

Wrong Value

A wrong value can be a data-flow problem (e.g., a variable value) or a control-flow problem (e.g., a wrong call stack structure). In this case, the developer directly attempts to locate the point where the wrong value is created. For example, if the execution point is buggy because the `foo` value is wrong, the developer tries to locate the execution point where this wrong value was set.

In breakpoint-based and log-based debugging, developers have to guess the possible origins and check the correctness of their guesses before they can finally locate the desired execution points.

Missing Value

In this case, developers have to first answer this question: "Where do they expect the value to be created in the buggy execution?" A simple answer to this question is a source code location. Then they need to check for the execution point reachability, as we explain in the next section. If the execution point happens in the buggy execution then they need to inspect the point in order to understand why the value is not created (e.g., the Java class is not loaded), or if it is created, what the execution points in-between are that remove the value. In the case where the execution point does not happen in the buggy execution, developers need to check reachability for the execution points that are expected to happen before this point until they find one that happens in the buggy execution.

3.7.3 Search

The above approaches are appropriate if the developer knows a correct execution point close enough to the fault or a buggy execution point that can be used for backwards navigation. But how do they select the execution point when they start inspecting buggy execution and no determined execution point exists, or when they feel blocked with the above approaches and need to start from a fresh execution point?

In these situations, developers often attempt to search in the suspicious interval with the hope of locating new correct or buggy execution points. A simple search, such as a line breakpoint, can result in all execution points that occur in a specific line. A more comprehensive search might require particular values at the execution point (e.g., conditional breakpoints), or a specific sequence of execution points.

In order to conduct a search, the developer has to answer two main questions: What are the search criteria? How can these search criteria be translated to the debugger concepts? A search result usually contains more than one execution point; therefore the developer has to iterate over the result execution points to identify the useful ones. To get less irrelevant execution points in the search result, they could refine the search after a few execution point inspections with additional criteria. Here, we discuss two common scenarios that developers need to search the execution:

From Bug Symptoms to a Buggy Execution Point

When the developer starts debugging, there is no inspected execution point. The developer needs a point to start inspection. Of course, the developer can start from the beginning of the buggy execution, but it might not be the most efficient way of selecting the first point. A good candidate for the first execution point is the place where the bug symptoms appear. Some bug symptoms such as exceptions or error messages usually provide data (e.g., the error line number) that helps the developer identify the execution point. For example, the developer can set a breakpoint that halts the execution whenever the exception occurs.

Unfortunately, selecting the buggy point where bug symptoms appear is not always easy. For other types of bug symptoms, such as wrong output values, issues in the graphical user interface, execution slow down, and so on, developers might have a hard time locating the corresponding buggy point. They have to guess the relevant code or conditions of the bug symptoms and then search over the buggy execution. A general form of this problem appears when the developer needs to relate an external event or value to the program internal state.

Execution Point Reachability

One of the common questions raised in the locating-defects process is whether a particular point happens or is reached in the buggy execution. For example, "Is method `foo` called when user opens an e-mail?" or "Does the execution reach this line before the error?" When the answer to this question is "no", the developer needs to know where the execution deviates from the expected path. Hence, the developer forms new reachability questions for the preceding points in the expected execution until they find one of these points in the buggy execution.

3.7.4 Issues

We discuss three common issues appear in the above approaches.

Discontinuous Call Stack

One of the common issues in the controlled forward navigation appears where there is a disconnection or fork in the logical process that the developer follows. A common case happens in asynchronous calls where the task is executed with a delay or in another thread. Another case happens when the logical process goes out of the debuggee (e.g., to another program or to a low-level component). In both cases, developers need to jump to the next logical point.

Missing Cause and Effect Chain

In the causality backward navigation, it usually happens that parts of the cause and effect chain is not in the debuggee. For example, suppose that the debuggee is a client application. This client sends messages to and receives them from a server. At some point a wrong value appears in the client. The developer finds out that this wrong value comes from the server. As the server code is trusted, it is very likely that the wrong value is created due to issues in one of the messages sent from the client to the server. In order to find the erroneous message, the developer either has to trace back the wrong value in the server or check all the messages sent from the client to the server.

Multiple Instances of Executions

So far we have assumed that the developer inspects one instance of buggy execution, and we have not discussed issues caused due to inspecting multiple instances of buggy execution. However, the navigation on the buggy execution is one-way and the developer can not freely move on it. Therefore, whenever developers need more data at one of the past execution points, they need to re-execute. The main issue with re-execution is that the developers need to remap their mental model to the new execution and restore the previous debugging state.

3.8 Changes in the Expectations

As we mentioned earlier, the set of hypotheses that forms the expectations might change during debugging. What are the implications of this fact on the structure of process and strategies used by developers? After every change in the expectations, the developer needs to review and update the statuses of previously inspected points. As a consequence the suspicious interval might change. After updating the suspicious interval, the developer could continue the process with the same or a different strategy based on the new expectations. In summary, the general structure of the process of locating defects does not change due to changes in the expectations, however the developer has to update the debugging state after every change.

3.9 The Roadmap of Improving Debuggers

The above analysis and discussion lead us to several potential improvements in debuggers.

3.9.1 Supporting the Basic Concepts

Execution Point

The structure of process of locating defects tells us that the developer's mental model of the buggy execution consists of the set of inspected execution points, their relations, and whether these execution points are correct, buggy, suspicious, or none. A debugger should save the inspected execution points and let the developer look at them again if needed. For example, a breakpoint debugger should let the developer go backward on the inspected execution points though it can not move the execution back to those points.

Suspicious Interval

It is obvious that if the debugger recognizes and supports the suspicious interval, it can automatically limit the subsequent point selection to this interval and therefore not only select the execution point(s) faster but also relieve the developer from excluding many irrelevant execution points.

The suspicious interval is not only useful for limiting the search space for the subsequent execution point selection, but it also provides an insight into the inspected execution points presentation. First, a time line that highlights the suspicious interval and shows the order of inspected execution points, can increase the developers' awareness of their search. Second, although developers should be able to access all inspected execution points, they may not be interested in seeing all of them. Perhaps a summary that explains how the developer is lead to the current suspicious interval is sufficient. For example, in Figure 3.6, once the developer moves to the third suspicious interval, almost all inspected points in the interval from B to A (except C) are not important anymore to the developer and can be automatically cleaned from the debugger interface.

Expectations Changes

After every change in the expectations, the developer needs to update the execution point statuses, remove irrelevant ones, update the suspicious interval. In breakpoint-based and log-based debugging, developers have to manually remove the unnecessary breakpoints and log statements. A debugger should support updating the debugging state with a little effort from the developer.

3.9.2 Supporting Natural Ways of Execution Point Selection

A debugging approach, and in particular a debugger, can greatly facilitate the developer's job if it provides more natural ways of selecting execution points that are closer to the queries in the mind of developers. It can also remove the burden of obtaining the desired/suspicious parts of program state by identifying and presenting them to the developer.

Controlled Forwards Navigation

We can imagine that debuggers provide developers with new ways of defining steps and summaries (e.g., a developer can start from the current source code location and moves over the control-flow graph to select lines and also data to be collected and the final destination). Debuggers can also remember and let developers reuse steps later from the same location. For example, in an e-mail web client, the stored steps after user login can be `toNextUserAction`, `toOpenAnEmail`, `toComposeSendButtonPush`, etc.

It is even conceivable that debuggers suggest the next step or automatically perform the developer's checks in every step. For example, the suggestion can be achieved by static or dynamic code analysis and identifying the waypoints. For the automated checks, debuggers can provide developers with means (e.g., by writing assertions) to describe the conditions that must be checked at every step. The main difference between these assertions and traditional assertions inserted in a source code is that they have access to the preceding point data and the step summary, which makes them more expressive.

Causality Backwards Navigation

Debuggers can simplify developers' job by providing functionalities that automatically locate the origins to a wrong value for developers. For example queries like: "What is the last execution point that this property value was set", or "What is the origin to this value?"

Search

Of course, a debugging approach that supports common queries by default (i.e., makes answering to the second question needless) can greatly facilitate developers' job.

The general form of this problem appears in the boundaries of the debuggee where it sends and receives data. Debuggers can help developers in this regard in two ways; first, they let developers debug all programs (e.g., client and server) and connect send and receive functions in these programs. Second, they provide special queries for selecting boundary execution points (e.g., the execution points where client sends, receives data to the server).

3.10 Related Work

Researchers have conducted several studies to understand how developers comprehend programs. Most of these studies tried to provide models of program comprehension process. Although these studies suggest different models, their findings are greatly consistent with each other. Several studies suggest that developers begin with questions and form hypotheses.

Then they try to test their hypotheses [Brooks 1972, Vans 1999].

Other models focus on high-level strategic differences, such as whether developers understand programs from the top-down or bottom-up [Corritore 2001, Littmann 1986, Vans 1999]. A recent study describes similar results, framing program understanding tasks as“fact finding” missions, driven by developers’ efforts to discover properties of the program at varying levels of abstraction [LaToza 2007].

Araki et al. described “a general framework for debugging” [Araki1991]. The main elements of framework are hypothesis selection and verification. Although the framework does not explain how developers inspect the buggy execution, it describes two general strategy for hypothesis selection: narrowing the suspicious region and expanding the certified region. Agrawal et al. analysis on causality backward navigation, which is used as a basis for building SPYDER, is very similar to ours [Agrawal1993].

Ko et al. study program comprehension in debugging tasks. They describe that developers search for relevant focus points to investigate, relate information to these points by investigating neighboring statements or methods [Ko2008]. Our analysis is in conformance with previous studies but reveals more details about the debugging process, developers’ mental model and decisions, which can be used to improve debugging tools and techniques.

3.11 Summary

In this chapter, we have analyzed the locating-defects process and answered several important questions. We have modeled the developers’ mental model as two sets of hypotheses and facts. We have shown that inspecting buggy execution can be summarized as a sequence of execution point inspections. Developers limit their search for the subsequent execution point to the suspicious interval, which starts from a correct and ends at a buggy execution point. They employ three different general approaches, controlled forwards navigation, causality backwards navigation and search. We have also mentioned several issues such as “discontinuous call stack”, “missing cause and effect chain”, and “multiple instances of executions”, which appear in these approaches. Based on the results of this analysis, we have suggested several improvements and features in debuggers.

Do not follow where the path may lead.
Go instead where there is no path and
leave a trail.
— Harold R. McAlindon

4 Querypoint Debugging

According to the model introduced for locating-defects process in chapter 3, execution points play a key role in the debugging process. However, what a developer understands from an execution point is not as limited as the definition we provided in section 3.1, which is static and inflexible. Instead, a developer builds up an abstract dynamic definition for each inspected execution point based on its features such as the program state values, the way it affects the execution, how it is affected by other inspected execution points, etc. *Querypoint debugging* is about enabling debuggers to store, understand and process these abstract definitions in an efficient way.

In the widely-used approaches, breakpoint-based and log-based debugging, a developer has to describe the execution points in terms of indirect values, e.g., line numbers. These indirect values are usually far from the definitions the developer has in mind : they are static and there is almost no connection to other inspected execution points values. Re-executions make the gap even larger. A developer loses the debugging state after every re-execution, because the debugger is not able to restore it on the new instance of buggy execution.

As we described in chapter 2, two new debugging approaches try to address this issue. One approach, *instruction/event indexing*, assures that all re-executions are identical. So, the execution points can be re-located on a new execution by their indexes in the previous execution. *Capture-replay*, the other approach, attempts to solve the problem by not re-executing : enough data is captured from one execution in order to relieve the developer from re-execution. So, there is only one execution and one instance of each execution point, then no need for re-locating execution points.

We pursue a totally different approach to tackle the problem: no constraint is imposed on the re-execution and developers can still use their ordinary test-cases for debugging. Instead, we try to expand the execution point definition as developers do. Execution points are abstracted away from one execution to a set of executions. Therefore, they can be re-located on the subsequent re-executions by querying their abstract definitions. In this approach, the debugger's focus (the subsequent execution point) is selected by the developer; then the developer is informed by the collected data at the execution point; then the developer re-focuses the debugger on another point.

In this chapter, we introduce *querypoint debugging*, as a novel approach to debugging, which centralizes the *querypoint* concept in the debugging process. A *querypoint* is the abstract definition of an execution point from a developer's perspective based on *why* or *how* it is visited. It also contains information about the *inspected values*, *the expected values or the assertions that must hold* and the *outcome of inspection*.

We start with an introductory example, which provides a general idea of this approach. Then, we explain how different types of execution point selection, *causality backward navigation*, *search* and *controlled forward navigation*, can be abstracted from one execution and re-located on the subsequent re-executions. Finally, we demonstrate how the *querypoint* approach can provide solutions to many problems of the traditional debugging approaches listed in section 2.1.3.

4.1 Introductory Example

Figure 4.1 demonstrates a buggy Java program[1]. The program processes a list in two consecutive loops and computes and sets new values for each item in the list. Every item in the list has a boolean field with name `bar`. This value is set in the first loop (line 7) and used in the `if`-condition in the second loop (line 18). The bug happens when the program throws an `AssertionError` exception in the second loop (line 24). We chose two different strategies, backward and forward debugging, that might be used to debug this code, in order to demonstrate how querypoints are used in practice.

Backward Debugging

With this strategy, a developer starts from the bug symptoms and traces them back to the fault. The first execution point that the developer looks at is where the bug symptoms appear, in our case where the exception occurs. The corresponding querypoint **A** to this execution point is "where `AssertionError` occurs." This definition is not dependent on any specific execution and identifies the execution point where the bug symptom appears on all re-executions. For the moment, assume that only one such execution point exists, i.e., only one `AssertionError` occurs in the execution. We will discuss this issue later.

The debugger re-executes the program and locates the execution point corresponding to querypoint **A**. The developer does not expect `foo` to be `null`. The inspection outcome is an assertion for querypoint **A** that "`foo` must not be `null`." In order to understand how this `null` value was set, the developer has to find the execution point where this value was set. The corresponding querypoint **B** is "the execution point where the last write to `foo` of point **A**

[1]This example resembles a real case; however it is simplified for presentation purposes.

```
1   //-------- the first loop --------
2       for (Object record : list){
3           record.bar = true;                          // D
4           try{
5               stmt1;
6               stmt2;                                  // E
7               record.bar = expr1;
8               stmt3;
9           }catch(Exception exp){
10              //ignore the exception
11          }
12      }
13
14      Object foo;
15
16  //-------- the second loop --------
17      for (Object record : list) {
18          if (record.bar || cond) {                   // C
19              record.bar = false;
20              foo = expr2;                            // B
21          } else{
22              foo = expr3;
23          }
24          assert (foo != null);                       // A
25      }
```

Figure 4.1: A Java pseudo-code which processes a list in two consecutive loops. The comments at the end of lines give the name of querypoints discussed in the backward strategy.

occurs." Again this definition is independent from any specific execution.

From the source code it is obvious that the last write to `foo` must happen at line 20 or 22. Suppose that it happens at line 20. Again the developer thinks that the execution must have gone through the other branch. The developer is suspicious about the `if`-condition at line 18. So, the inspection outcome is "wrong branch is taken from `if`-condition at line 20". The next querypoint, **C**, is " the execution point where line 18 is executed just before point **B**" or "the last condition brings the execution to point **B**".

After inspecting the values at point **C**, the developer realizes that `record.bar` is `true` which is incorrect. So again, the inspection outcome for this querypoint is "`record.bar` must not be `true`". The next natural querypoint, **D**, is "the execution point where `record.bar` of point **C** is set."

The debugger re-executes the program and locates the next execution point (**D**); it happens at line 3 though the developer expects line 7. It seems that something prevented the execution from reaching line 7. The outcome of inspection of point **D** is "the last write to `record.bar` must not occur at line 3, but after that (line 7)." The developer has a guess : an exception

QP	Abstract Definition		Inspected Value(s) / Step Summary	Assertions / Inspection Outcome
Backward Debugging				
A	The execution point	where AssertionError occurs. [**Search**]	line 24, foo value which is null	foo value must not be null
B		where the last write to foo of point **A** occurs. [**Causality Backward : Wrong Value : Data**]	line 20	The wrong branch is taken from if-condition at line 20
C		where line 18 is executed just before point **B**. [**Causality Backward : Wrong Value : Control**]	line 18, record.bar value which is true	record.bar value must not be true
D		where record.bar of point **C** is set. [**Causality Backward : Wrong Value : Data**]	line 3	The last write to record.bar must not occur at line 3, but after that (line 7)
E		where an Exception occurs, after point **D** and before point **C**. [**Search**]	line 6	This is **the fault**.
Forward Debugging				
S	The execution point	where execution reaches line 2 for the first time. [**Search**]	line 2	
T[i]		where i*th* step **T**, which is a loop iteration, after point **S** ends. [**Controlled Forward**]	The **T[i]** step summary, which contains record.bar final value and all exceptions occurs in the step	No exception and record.bar value was set appropriately.
U		where is a T[i] and the **T** assertions fail. [**Search**]	line 6	This is **the fault**.

Table 4.1: This table contains the list of querypoints defined in the introductory example. The first column (QP) is the name of querypoint, the second column (Abstract Definition) is its abstract definition, the third column (Inspected Value(s)) shows the values inspected by the developer and finally the last column (Assertions) is what is asserted by the developer and their result.

occurs at lines 5 or 6 which prevents line 7 execution. The next querypoint, **E**, is "the execution point after point **D** and before point **C** where an Exception occurs." Similar to querypoint **A**, assume that only one exception occurs in this interval.

The debugger locates point **E** which occurs at line 6. The developer finally identifies this point as the fault. The sequence of querypoints **E** to **A** completely explains the diagnosis process. Table 4.1 shows all 5 querypoints and the information they contain.

Forward Debugging

With a forward strategy, a developer starts from a correct execution point, for example the beginning of the first loop and follows the execution until the fault, the first execution point where something goes wrong, is identified. The first querypoint, **S**, locates the beginning of the first loop. From this point, the developer decides to use *controlled forward navigation*. As we explained in the previous chapter, in this navigation the developer chooses a *step* for moving forward.

In this case, we assume that the developer chooses a loop itereation of the first loop as the first step. Let us call this step **T**. So the next querypoint, **T[1]**, is where this step ends, which locates the execution point just before the next iteration of the loop or just before the execution exits from the loop. Therefore, **T[1]** can locate different instructions depending on how this step ends.

After every step, the developer looks at the *step summary* and determines whether there is any problem. A step summary contains the important program events and state changes. A developer can define the step summary for a step, by specifying the events or values that must be collected, or using a default step summary, which collects data at each instruction within the loop. We can assume that the step summary for step **T** contains "`record.bar` final value and all exceptions occur in the step".

The developer can re-use step **T** to check step summaries at **T[2]**, **T[3]**, etc. However, it requires that the developer checks each step summary, and if the loop has many iterations it might take a long time. In order to mitigate this problem, the developer can define **step assertions**, which can check constraints on the step summary. For example, in this case, the developer can write assertions that make sure that no exception occur and `record.bar` value is set appropriately.

After defining the step assertions, the developer can define a new querypoint **U**, which identifies the fault. **U**'s definition is "starting from point **S**, move forward with **T** steps until the assertions fail or no more **T** step can be taken (i.e., it is the end of the loop).

4.2 Querypoint Types and Definitions

In this section, we describe three types of querypoints for a Java like language, corresponding to the three main approaches for reducing the suspicious interval : *causality backward navigation*, *search* and *controlled forward navigation*.

4.2.1 Causality Backward Querypoint

Based on our discussion in 3.7.2, causality backward navigation, can be performed due two types of problems, wrong data—which can be a wrong alias value or expression (e.g., an addition or a method call) value—and control values.

lastChange is the querypoint that identifies the execution point where the alias value was set. *lastChange* needs three parameters : the execution point where it is queried, here it is **A**, the alias and the wrong value. Keeping the wrong value allows the debugger to check it on every new execution and makes sure that it still holds. If the value is an object, a shallow copy of the object or only the property values that are inspected by the developer should be stored. For example, in the introductory example, we can re-write point **B** in this form: `lastChange(A, foo, null)`.

valueOrigin is the querypoint that identifies the execution point where the expression value is created. For example, if the expression is an addition, *valueOrigin* gives the execution point just before the addition, so the values of two operands are collected. Or, if the expression is a method call, *valueOrigin* locates the execution point where the method returns.

Wrong Control Value

If there is a problem in the control-flow, it means that the current thread should not have been started or one of the branches in the current call stack should not have been taken. *lastCondition* identifies the execution point where the wrong condition is executed for the last time. For example, in the introductory example, we can re-write point **C** in this form: `lastCondition(B, "line 18", 1)`. In practice, we need more detailed information for identifying the condition such as the file name and instruction id. Value `1` in the definition tells us that the first branch was taken.

4.2.2 Search

A search querypoint consists of a query that selects a set of execution points and an index that identifies one of these execution points. The query can be defined in any language or means (e.g., ordinary or graphical breakpoints, aspect oriented join points, etc.) that is more appropriate for the developer. In order to support *suspicious interval* a search querypoint can be bound to an interval specified by two other execution points. Moreover, a search querypoints can use the values in the previously inspected execution points. For example, suppose a line breakpoint that selects all execution points where a specific line is executed. The condition “`bar==inspectedValue(A, foo)`” on this breakpoint assures that the execution point is selected if the `bar` value at the execution point is equal to the `foo` value at point **A**.

4.2.3 Controlled Forward Navigation

A developer uses this approach to monitor a data-flow or a collection of data-flows. However, in a language like Java, data-flows are not explicitly defined. Here, to simplify our discussion we reduce the data-flow concept to the thread concept, because a thread usually represents a data-flow. We discuss this issue in 8.1.2.

A controlled forward querypoint consists of a *step* definition, its *scope*, and its *summary*. A *step* definition is similar to a search querypoint, which selects the execution point at the end of *step*. However, it is limited to the current thread (or data-flow). The step *scope* limits the *step* with another execution point, for example, the return of the current method. The step scope gives a guarantee to a developer that the subsequent execution point is not very far from the current execution point. For example, in the case of occurring an error during the step execution which prevents a normal halt at the end of the step, the execution do not jump out of the current method. The step *summary* specifies the data must be collected in the step. In addition to the above elements, a controlled forward querypoint can contain an *assertion* element, which checks the step summary.

4.3 Querypoint Processing

Identifying the execution point corresponding to a querypoint on the buggy execution is called *processing* the querypoint.

4.3.1 Execution Point Selection

A querypoint is transformed to a set of constraints on execution points, and these constraints are used for selecting the right execution point.

Constraints on Instructions

The first group of constraints limit the Querypoint to execution points with specific kinds of instructions. We call them the querypoint *potential* instructions. For example, a line breakpoint has one potential instruction : the first instruction in the line. Or, the potential instructions of a lastChange querypoint are the instructions that could change the value. In the case of lastChange on a variable, only the variable assignment instructions in the block or scope that the variable is defined, and in the case of an object property, only the instructions that set a property with the same name are potential instructions. These constraints allow us to exclude most irrelevant execution points before the execution.

Constraints on the Process, Thread and Interval

These constraints limit the process, thread or interval that the execution point could occur. For example, “the thread object class is `EventDispatchThread`”, or “the execution point occurs after point **A** and before point **B**” are two such constraints. The thread constraints appear in the step definitions for limiting the subsequent execution point to the current thread (data-flow). The interval constraints are used for limiting the search to the suspicious interval.

Constraints on Runtime Values

The constraints require runtime values at the execution point classified in this group. For example, in order to check a condition like “`foo==0`” that is set on a breakpoint the `foo` value at the execution point is needed. These constraints are instrumented around the instructions selected by the above constraints. If the the querypoint is limited to an interval, for example it happens after point **A**, then the instrumentation is postponed until visiting and identifying point **A**. If the runtime constraints hold at the execution point, a tracepoint is captured. The tracepoints captured for a querypoint are called the *potential results* of querypoint.

Constraints on Two or More Execution Points

This group of constraints are defined dependent on values at other execution points. For example, the constraint "`bar==inspectedValue(A, foo)`" specifies that the `bar` value at the execution point must be equal to the `foo` value at point **A**. Note that there is no cycle in the querypoints dependent constraints because querypoints are defined one by one by a developer and, a new querypoint can be dependent only on already defined querypoints.

Two cases can happen in checking these constraints : (1) Point **A** is visited and the `foo` value is known, so the condition is checked like a constraint on runtime values, (2) Point **A** is not yet reached and the `foo` value is unknown, so the `bar` value must be captured at all instructions selected with the above constraints. When point **A** is identified and the `foo` value is obtained, the potential results (the captured tracepoints) are enumerated and checked against the condition. The tracepoints that do not satisfy the constraints are removed from the potential results.

We employ unique identities for connecting two values at two different execution points. For example, two object properties with name `foo` are considered one property if their owner objects have the same identity. Or, two variables with name `foo` are considered one variable if their scopes have the same identity. The identities could be provided by the underlying system, the debuggee program, or by instrumentation.

Index Constraints

An index constraint specifies a tracepoint in the potential results by an index. If the index constraint is not specified the first tracepoint in the potential results is selected by default. For accessing the tracepoints at the end of list of results negative indexes can be used. For example, for selecting the last one, index `-1` is used.

Inferred Constraints

The result of processing a set of querypoints on an execution is a trace that consists of a list of tracepoints, each one corresponding to a querypoint. So, after every re-execution a new trace is collected. The list of traces captured from all re-execution, the *reproduction history*, is kept by the debugger. We explain how the reproduction history is used for inferring new constraints and therefore optimizing the querypoints processing.

In addition to the constraints that are directly determined from the querypoint definition, we use the reproduction history to infer new constraints. It means that we assume that some values or events in the previous execution will happen in the same way in the new execution. For example, consider querypoint **B** with the definition "`lastChange(A, this.foo, null)`". If we have the tracepoint collected at point **A** in the previous execution and, it contains the `this` class name, we can use this information to infer a new constraint for querypoint **B** : the object contains property `foo` must be an instance of the same class. Here, we made a delicate assumption : the `this` object class will not change in the new execution.

Another type of constraints that can be inferred from the reproduction history is the index of potential execution point which became the querypoint result. Suppose that the first time we processed a querypoint, we had to collect 1000 tracepoints and the result was the 500th tracepoint. If the same numbers happen in several executions then it is very likely that they happen again. We can use these numbers to infer an indirect constraints like a hitcount constraint. Then, there is no need for data collection at other execution points, we only collect data at the 500th hit.

The danger of the inferred constraints from the reproduction history analysis is that the assumptions behind them might not be true in a new execution. It is important to be able to check assumptions. For example, in the case of inferred object class, the `this` class at point **A** must be checked to be the same as previous executions. But, what if it is not? The constraint is extended with an OR : the object is type one OR type two. Or, the inferred constraint is dropped.

4.3.2 Data Collection

The result of processing a querypoint is a tracepoint that contains collected data at the execution point. The data consists of *general*, *querypoint*, *dependent*, and *user* parts. The *general* part is collected for all tracepoints similarly and contains data about the process (e.g., process id) and threads (e.g., call stack). The *querypoint* part contains querypoint-specific data. For example, in the case of *lastChange* querypoint the old and new values are collected. The *dependent* part is the data that is required for computing other querypoints that are dependent on this querypoint. The *user* part is the data requested by the user to be collected at the execution point.

4.3.3 Pausing at Querypoints

If the result of processing of a querypoint could be a live execution point (like a breakpoint hit) instead of a tracepoint, we call the querypoint *pauseable*, otherwise *un-pauseable*. Two groups of querypoints are pauseable. First, querypoints that are pauseable by definition : they are not dependent to any execution point that occurs after them. Second, querypoints that can become *pausable* by inferred constraints. For example, an inferred index constraint can be used for halting the execution at the querypoint result.

4.3.4 Inconsistencies

Suppose that querypoint **B** is "`lastChange(A, this.foo, 0)`". What does happen if in the new execution at point A, `this.foo` value is not `0` (e.g., `1`) or, `this` object has no property with name `foo`? The initial solution is informing the developer about the inconsistency. But, how does the developer continue debugging?

An inconsistency is either merged to the previous definition or, causes debugger to consider a new different class of executions. For example, in the case of value change, the new value is merged in the **B**'s definition, so we have "`lastChange(A, this.foo, {0,1})`". But, in the case of missing property, a new class of executions are considered : the executions that `this` has not property `foo` at point A. The reason that we split in this case is that the subsequent execution point selected after **A** .

4.3.5 Performance

A natural way to compare the performance of this approach to other debugging approaches is by measuring or estimating the total time needed for debugging a bug. However, a developer is not busy in all the debugging time. For example, after defining a new querypoint, a developer has to wait until the debugger selects and shows the result execution point. In this time interval, the developer can work on another task. Therefore, the total debugging time can be split to *active* and *inactive* times. In querypoint debugging, the active time is mainly consists of inspecting execution points and defining querypoints. The rest of debugging time is inactive: the debugger processes querypoints and the developer can work on another task until the result is ready.

The inactive time is directly related to the time taken to process a querypoint. Because processing a querypoint is carried out by monitoring live executions, the processing time is dependent on two parameters. First, the execution time until identifying the result, which is less than or equal to a reproduction cycle. Consider that for forward and search querypoints we do not need to wait a full reproduction cycle. Second, the overhead incurred by data collection needed for processing the querypoint. The execution overhead is dependent on the number of execution points in which data is collected and the amount of data should be collected at each execution point.

Comparing to widely-used approaches, log-based and breakpoint-based debugging, querypoint debugging removes the hard task of setting breakpoints and inserting log statements, which usually takes a huge amount of debugging time. Moreover, in these two approaches, almost all debugging time is active. A developer has to check breakpoint hits and monitor log records during execution.

Comparing to new approaches, instruction indexing and capture-replay, the active time is pretty the same but the inactive time pattern is different. Both approaches need a considerable time—at least 5 (in the first approach) and 22 (in the second approach) normal reproduction cycles—for capturing enough data from the first execution. The rest of inactive time, similar to querypoint debugging, consists of query processing. In the first approach, a query is processed by deterministic re-execution, which incurs at least 15 times execution overhead. In the second approach, the query is applied on a database, which contains the full execution trace.

Suppose a buggy execution which takes about 30 seconds and a developer requires the answer to 10 querypoints (or queries) to locate the defects. Suppose that processing each querypoint (or query) incurs 3 seconds overhead. So the total inactive time in querypoint debugging is $30 + (30 + 1 \times 3) + \ldots + (30 + 10 \times 3)$, which equals to 465 seconds. The inactive time of the instruction indexing approach is $30 \times 5 + (30 \times 15 + 1 \times 3) + \ldots + (30 \times 15 + 10 \times 3)$, which equals to 4665 seconds. The inactive time of the capture-replay approach is $30 \times 22 + 3 \times 10$, which equals to 690 seconds.

4.4 What Does It Improve?

In 2.4, we named several issues exist in widely-used debugging approaches. Here, we explain how querypoint debugging provides direct or indirect solutions to those problems.

Manual Re-execution

Querypoint debugging does not directly solve this problem, however, because it relies on re-execution, it gives a good reason for tool developers to support test-case generation and also motivates developers to write test-cases for debugging sessions.

Prolonged Execution

Again, querypoint debugging does not directly attack this problem, however, the debugger can use the suspicious interval for employing shorter reproduction cycles.

Lost Debugging State

This problem is solved : the main feature of querypoints is that they can be re-located on a new buggy execution. So, a developer can continue an execution point inspection after re-execution.

Guesswork

The guesswork is significantly reduced by providing an expressive querying mechanism which allow developers to refer to previously inspected values.

Irrelevant Data

A developer only expresses what is needed to be inspected by a querypoint, and the debugger has to find the execution point. There is no intermediate effort for the developer.

Overloaded Developer Memory

All steps and their outcomes are stored in querypoints, so a developer has not to memorize them and, can simply obtain them through the debugger interface.

Backward Navigation

Backward navigation is enabled by re-execution and identifying the queried execution point on the new execution.

Multi-Environment Support

Querypoints can solve part of the problem : connecting execution points in two different environment without capturing a full trace; The same identifier technique works here for connecting an execution point which sends a message to another execution point which receives the message.

Diagnosis Integrity Check

Employing querypoints, a debugger is able to assure that the values at the inspected execution points in the new execution is consistent with the previous executions.

Debugging Artifacts

Querypoints defined in a debugging session can be stored as the debugging artifacts. They contain enough information for restoring the debugging session.

Every failure is a step to success.
— William Whewell

5 A Querypoint Debugger

In this chapter, we will explain the design and architecture of a querypoint debugger that employs a generic trace querying mechanism for the querypoint implementation. A trace query, or a *tracequery*, selects a set of execution points from an execution. So, the result of processing a tracequery on an execution is an array of tracepoints. A querypoint in this debugger is defined by a tracequery and an index, positive from the beginning, or negative from the end; for example, index zero (0) selects the first tracepoint and index minus one (-1) selects the last tracepoint in the tracequery result. So, the output of processing a querypoint on an execution is at most one tracepoint.

As we demonstrate in this chapter, this structure, a tracequery and an index, used for defining querypoints, allows us to define three types of querypoints : controlled forward, search, and causality backward. However, in some cases it does not provide all features of a querypoint. for example, in the case of controlled forward, this structure is only able to define a step but not the step summary associated with a step.

In order to debug a buggy execution, the debugger requires a *debug configuration object*, which contains information about the debuggee, its running processes, instructions for reproducing and managing the buggy execution. A debug configuration object also includes tracequeries and querypoints previously defined for this buggy execution. In practice, the configuration data could be kept in an XML or a Java file, and therefore can directly be edited by a developer.

After loading the debug configuration, the debugger is ready for debugging the buggy execution. A developer can define new tracequeries or querypoints through the debugger interface and has the debugger to locate the results on the buggy execution. The debugger, then, reproduces and monitors the buggy execution and shows the results to the developer in its trace viewer.

5.1 Tracequery Definition

A tracequery consists of two parts. The first part, *the basic query*, is defined in a language that selects a set of execution points. The second part, *the script*, is written in BeanShell[1], that is evaluated at every execution point selected by the basic query and returns a boolean. The execution point is selected by the tracequery if the result is true. In fact, the basic query is the static part of a tracequery, whereas the script is the dynamic part. The static part allows a developer to benefit from optimizations in its processing, whereas the dynamic part allows a developer to exclude all irrelevant execution points employing a powerful language.

5.1.1 Basic Query

The language used for defining a basic query is based on selecting events by restricting their properties. Consider the following basic query :

```
methodentry{ callee{ clazz{ name{  *SunGraphics2D* }}} method{ void draw* }}
```

The following shows the above query in a tree like structure:

```
methodentry{ //constraints on a method entry event
    callee{ //constraints on the callee (the object that its method is called)
        clazz{ //constraints on the class of the callee
            name{ //constraints on the class name
                *SunGraphics2D*  }}}
    method{ //constraints on the method that is called
        void draw* }
}
```

The first token specifies an *event type*, method entry. This event type restricts the set of all execution points to only execution points that are method entries. After the event type a long brace appears, which contains properties that restrict the set of execution points even more. The first property, *callee*, defines restrictions on the object that contains the method. Its *clazz*'s, or class's, name must match `*SunGraphics2D*`. The next property restricts the *method*; it must match `void draw*`. In summary, the above basic query selects all execution points where something (e.g., point, line, circle) is drawn to the output graphic.

A query is defined by specifying an event type and a set of constraints on its properties. Depending on the type of property, different constraints can be defined. For example, if the property is an object, then constraints can be defined on its class and value. If the propery is a class, constraints can be defined on its name, parents, and chidren.

[1]BeanShell is a script language based on Java (www.BeanShell.org).

Table 5.1 contains different event types, which can be specified in the basic query. The second column shows the properties of each event type. These properties can be used to set more constraints on the event types selected. The constraint type of each property is indicated in the parentheses besides the property name. Table 5.2 shows different constraints types that can be included in a basic query. The second column in this table contains the sub-constrains of each constraint. For example, an *object* constraint, has *clazz* and *value* constraints, which set restrictions on the object's class and value.

Event Type	Properties
ObjectCreation	object (Object)
FieldChanged	object (Object), name (Atomic)
VariableChanged	object (Object), method (Atomic), line(Atomic) , name(Atomic)
MethodEntry	caller (Object), callee (Object) , method (Method)
MethodExit	caller (Object), callee (Object) , method (Method)
Exception	clazz (Atomic)
Line	object (Object), line (Atomic)

Table 5.1: Event types and their properties in the basic query language.

Constraint Type	Sub-constraints
Object	clazz (Class), value (atomic)
Class	name (Atomic), superclass(Atomic), subclass(Atomic)
ObjectRef	pointname (Atomic), frame (Atomic), reference (Atomic)
Method	clazz (Class), signature (Modifiers Returntype Name)
Atomic	data (String[])

Table 5.2: Constraint types and their sub-constraints in the basic query language.

5.1.2 Script

A BeanShell script is evaluated similar to the way a Java code is executed. However, in order to allow the script to access the execution point data and history of execution, a *tracepoint context object* under the alias *tracepointContext* is accessible by the script. This object allows the script to access live values at the current execution point and also values from previously selected execution points. For example, Figure 5.1 shows a script that is used with the above basic query in order to select only drawings with the black color.

The above script looks very lengthy for a simple color check. With employing a utility method provided by the tracepoint context object we have the following shorter script:

```
return tracepointContext.areEquals("this.foregroundColor", Color.BLACK);
```

```
try {
    BreakpointEvent event = (BreakpointEvent)tracepointContext.getEvent();
    ObjectReference graphicsObject = event.thread().frame(0).thisObject();
    Field foregroundColorField = graphicsObject.referenceType().fieldByName("foregroundColor");
    ObjectReference foregroundColor = (ObjectReference)or.getValue(foregroundColorField);
    Method getRGBMethod = foregroundColor.referenceType().methodsByName("getRGB").get(0);
    IntegerValue rgb = (IntegerValue)foregroundColor.invokeMethod(
                (event.thread(), m, new ArrayList(), ObjectReference.INVOKE_SINGLE_THREADED);
    if (rgb.intValue() == Color.BLACK.getRGB())
        return true;
} catch (Exception e) {
    System.err.println("An error occured in evaluating the tracequery script : ", e);
}
return false;
```

Figure 5.1: A BeanShell Script checks the drawing color; it must be black.

5.1.3 Referring to Previously Defined Execution Points

Our trace querying mechanism has a special feature which is necessary for supporting debugging queries and implementing querypoints. A developer can refer to previously defined tracequeries or querypoints and compare to their values. So, defining every new tracequery and querypoint increases the expressiveness of the querying mechanism. For this purpose, a label can be assigned to a tracequery and querypoint. These labels, then, can be used for referring to them or the tracepoints selected by them.

For a tracequery **T**, **T**[i] refers to the i*th* execution point selected by the tracequery **T**. So, **T**[0] refers to the first and **T**[-1] refers to the last execution point. A tracequery can also be limited to two execution points with `after` and `before` keywords. For example, this tracequery :

```
exception.after(A).before(B)
```

selects all exceptions occur after execution point **A** and before execution point **B**. This particular feature is very useful for restricting the subsequent execution point selection to the suspicious interval.

5.1.4 Global Object Reference

In order to refer to objects and variables in the heap at different execution points, we use *global object reference* with the following syntax:

```
executionpoint_reference(stack_frame_index):object_reference
```

For example, **P**(1):`x.y` refers to property `y` of variable or field `x` in the second frame (the newest stack frame is indexed zero) at point **P**. If the execution point reference is not specified, the current execution point is considered for obtaining the value. If the frame index is not specified, the frame with index zero is considered. If the object reference starts with a dot, it refers to an object accessible through the event. For example, if the event type is fieldchanged, the field's owner object is specified by `.owner`.

5.1.5 Multi-points Constraints

Using global object reference, we can define constraints on more than one execution point. For example, *equal* is a multi-points constraints that assures two or more values are the same. The constraint `equal(A:x,B:y)` is true if the value or object `x` at point **A** is as the same as the value or object `y` at point **B**.

5.2 Debugging a Painting Application

To demonstrate how the debugger works, we use a simple buggy painting environment introduced in [Ko2008]. This environment (Figure 5.2) contains a left pane, which is the painting controller, and a right white pane, which is the space where a user can draw graphics by pressing and releasing the mouse button. A user can select one of the available drawing modes (pencil, eraser or line) from the left pane. The color is also specified by moving the three sliders corresponding to the main three colors.

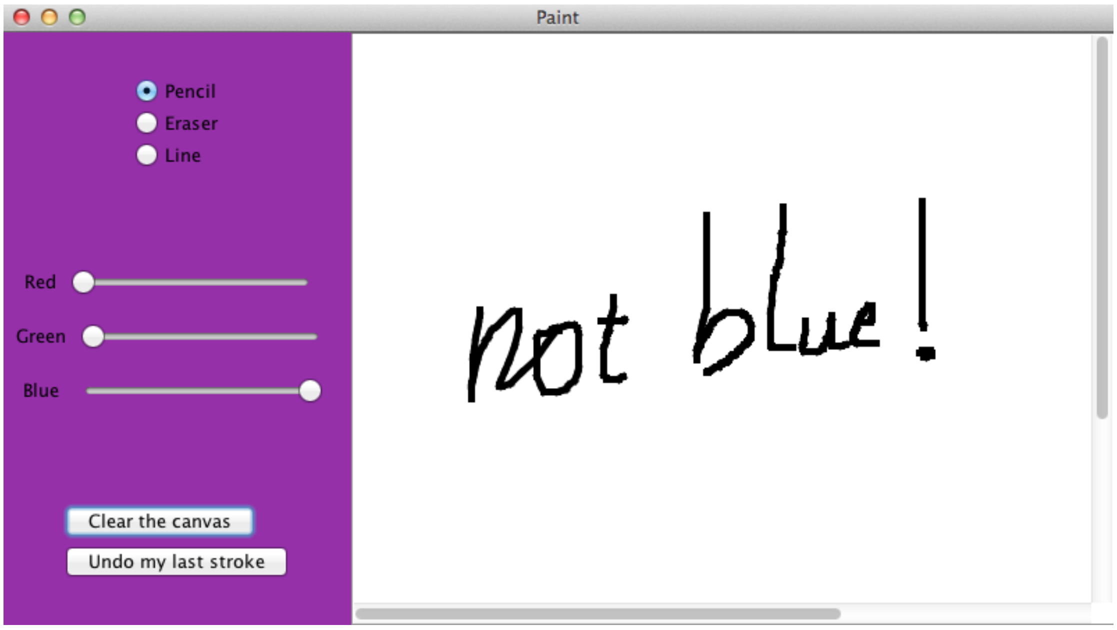


Figure 5.2: The buggy painting application

The bug happens when the user changes the color sliders' positions to draw a blue line (i.e., the red and green sliders are moved to the left ends whereas the blue slider is moved to the right end). In this configuration, new lines' color is black instead of blue (Figure 5.2). To reproduce the bug it is enough to re-start the program and move sliders to the same positions and draw a line.

In order to debug this program, we need to provide a debug configuration object to the debugger. Figure 5.3 demonstrates the `DebugConfig` interface and `PaintingApp_DebugConfig` class. In this class, a Java process with name `PaintingApp` is defined as a debuggee process. It also creates a `PaintingAppReproducer` object which is responsible for reproducing the buggy execution.

```
public interface DebugConfig{
    public List<DebuggeeProcessesDef> getDebuggeeProcessesDefs();
    public ExecutionReproducer getExecutionReproducer();
    public List<Tracequery> getTracequeries();
    public List<Querypoint> getQuerypoints();
    ...
}

public class PaintingApp_DebugConfig implements DebugConfig{

    ListDebuggeeProcessDef> debuggeeProcessesDefs;
    ExecutionReproducer executionReproducer;
    List<Tracequery> tracequeries;

    public init(){
        debuggeeProcessesDefs = new ArrayList<DebuggeeProcessDef>();
        debuggeeProcessesDefs.add(
            new DebuggeeeProcessesDef( "PaintingApp", DebuggeeProcessDef.TYPE_JAVA));
        executionReproducer = new PaintingAppReproducer();
        tracequeries = new ArrayList<Tracequery>();
        querypoints = new ArrayList<Querypoint>();
    }
    ...
}
```

Figure 5.3: The debugConfig interface, and its implementation, PaintingApp_DebugConfig class, which is used for creating the debug configuration object.

An appropriate strategy for debugging this application is starting from the bug symptoms and searching backward to the fault. In order to specify the execution point where the bug symptom, black line in the right pane, appears, we can use a tracequery similar to one we defined in the previous section. We can even change `draw*` to `drawLine` to reduce the number of selected execution points. This tracequery, call it **drawLine**, selects all execution points where a black line is drawn to the output graphic. Figure 5.4 demonstrates the result of the query in debugger trace viewer.

Figure 5.5 demonstrates how the data collected at a selected execution point can be viewed by selecting the execution point and clicking on "Print Info". A developer can also define a new querypoint by clicking on "Define a new point : ...". After defining a new querypoint, the debugger can locate and show the execution point corresponding to the querypoint in the subsequent executions (Figure 5.6).

If we look at the data collected at the first execution point, **drawLine[0]** or **P1**, we can identify

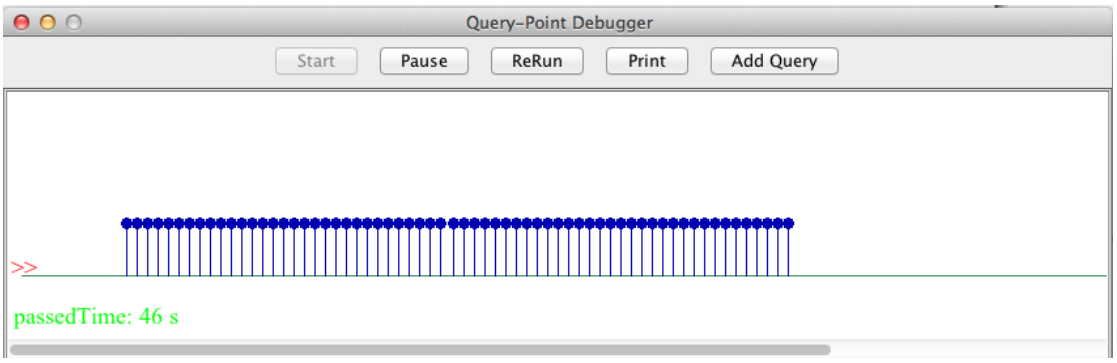
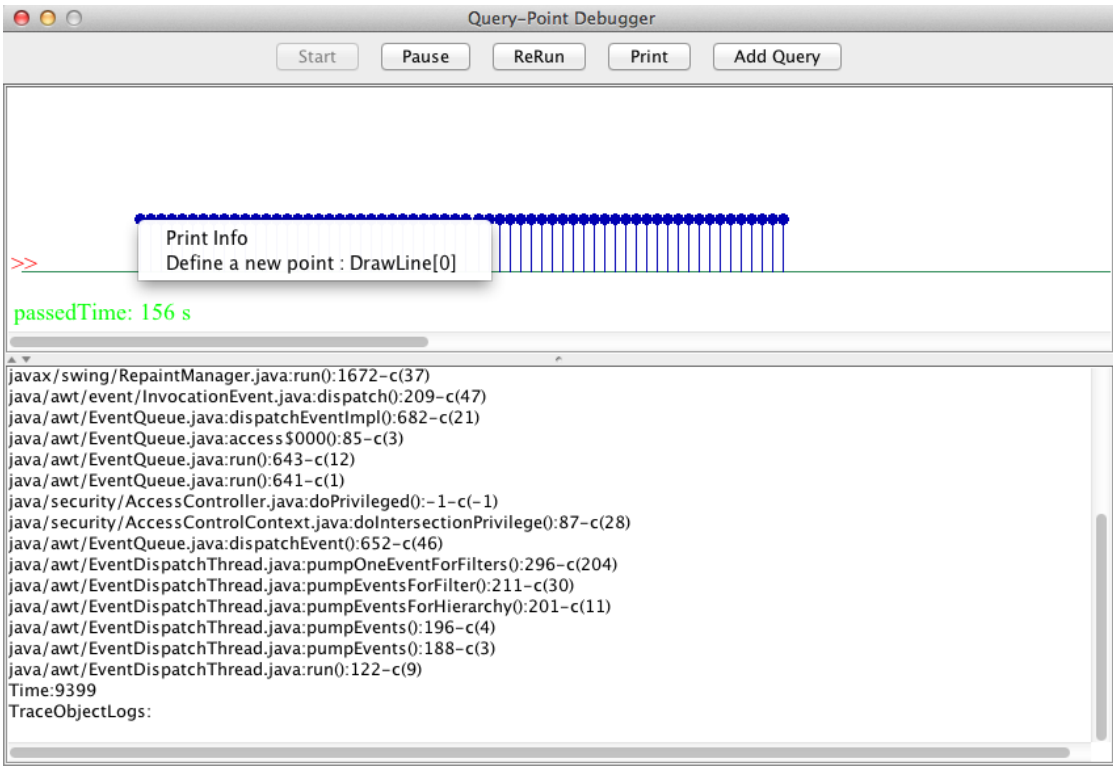


Figure 5.4: The screenshot of the debugger trace viewer after applying the tracequery that selects the execution points where a black line is drawn. Every blue circle shows an execution point.



```
javax/swing/RepaintManager.java:run():1672-c(37)
java/awt/event/InvocationEvent.java:dispatch():209-c(47)
java/awt/EventQueue.java:dispatchEventImpl():682-c(21)
java/awt/EventQueue.java:access$000():85-c(3)
java/awt/EventQueue.java:run():643-c(12)
java/awt/EventQueue.java:run():641-c(1)
java/security/AccessController.java:doPrivileged():-1-c(-1)
java/security/AccessControlContext.java:doIntersectionPrivilege():87-c(28)
java/awt/EventQueue.java:dispatchEvent():652-c(46)
java/awt/EventDispatchThread.java:pumpOneEventForFilters():296-c(204)
java/awt/EventDispatchThread.java:pumpEventsForFilter():211-c(30)
java/awt/EventDispatchThread.java:pumpEventsForHierarchy():201-c(11)
java/awt/EventDispatchThread.java:pumpEvents():196-c(4)
java/awt/EventDispatchThread.java:pumpEvents():188-c(3)
java/awt/EventDispatchThread.java:run():122-c(9)
Time:9399
TraceObjectLogs:
```

Figure 5.5: By right clicking on every execution point selected by a trace query, a developer can print the data collected at that execution point or define a new querypoint.

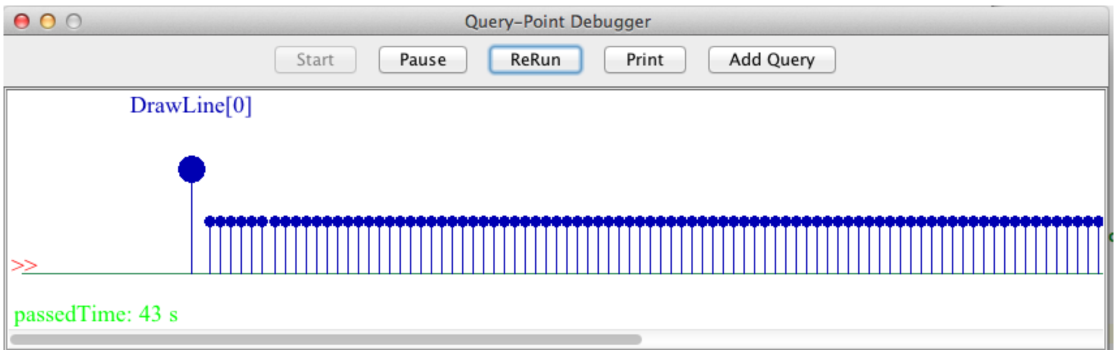


Figure 5.6: After defining a new querypoint, the debugger located it on a new buggy execution. The result is shown by a blue circle, larger than circles show other execution points.

the corresponding source code. As demonstrated in Figure 5.7, the call site of `drawLine` method is line 56 inside `paint` method in class `PencilPaint`. The next step is to understand how `g.foregroundColor` was set black.

```
----- Call Stack
    SunGraphics2D.java:drawLine():2098
    PencilPaint.java:paint():56
    PaintCanvas.java:paintComponent():42
    JComponent.java:paint():1027
    ...
    EventDispatchThread.java:run():122

---- Source Code
5   public class PencilPaint extends PaintObject {
...
47      public void paint(Graphics2D g) {
...
56          g.drawLine(one.getX(), one.getY(), two.getX(), two.getY());
...
60      }
61  }
```

Figure 5.7: Data collected at P1, the thread call stack and corresponding source code.

In order to find the location where g.foregroundColor was set, we define the following tracequery:

```
fieldchanged{name{foregroundColor}}.equal{.owner, P1:this}.before(P1)
```

This tracequery selects all execution points where the value of a property with name `foregroundColor` is changed, and the object owns the property is the same `this` object at **P1**, and they occur before **P1**. Call this tracequery, **foregroundColorChanges**. The next querypoint **P2** is **foregroundColorChanges[-1]** or the last execution point in the tracequery result. In fact, we are looking for `lastChange(P1:this.foregroundColor)` and we used our querying mechanism to implement the *lastChange* querypoint.

In order to process tracequery, the debugger reproduces the buggy execution and collects data at execution points where a `foregroundColor` property is changed. When the execution point reaches **P1**, the debugger checks the equal constratint by comparing object ids of `P1:this` and `.owner` in all selected execution points and filter out the irrelevant execution points. Finally, it demonstrates the last one as the result of querypoint. As demonstrated in Figure 5.8, line 50 in `PencilPaint` class is where the value is set. The object ids of `P1:this` and `P2:this` in the last execution were 1539.

```
----- Collected Data
P1:this <- ObjectId:1539, Class:SunGraphics2D
P2:.owner  <- ObjectId:1539, Class:SunGraphics2D

----- Call Stack
    SunGraphics2D.java:setColor():1653
    PencilPaint.java:paint():50
    PaintCanvas.java:paintComponent():42
    JComponent.java:paint():1027
    ...
    EventDispatchThread.java:run():122

---- Source Code
5  public class PencilPaint extends PaintObject {
...
47    public void paint(Graphics2D g) {
...
50      g.setColor(color);
...
56      g.drawLine((int) one.getX(), (int) one.getY(),
57                 (int)  two.getX(), (int) two.getY());
...
60    }
61 }
```

Figure 5.8: Data collected for identifying P2, the thread call stack and corresponding source code.

P1 and **P2** happen in `EventDispatch` thread. Whenever an event is fired which requires updating graphical interface this thread calls `repaint` method on the parent component and it recursively calls this method on children should be updated. The `repaint` method is called non-deterministically. So, object ids are not necessarily the same in different executions.

The next querypoint, **P3**, is `lastChange(P2(1):color)`. Similar to **P2**, we write the corresponding tracequery. Here, **P3** is dependent on **P2**, and **P2** is dependent on **P1**. So, in order to locate **P3**, the debugger collects data at all execution points that could be **P3** or **P2**. When the execution reaches **P1**, the debugger identifies **P2** and then **P3** among the selected execution points. **P3** occurs in line 60 of `PaintObjectConstructor` class (Figure 5.9).

After locating **P3**, developer seeks for the last change of the field `color` of `PaintObjectConstructor`. The next querypoint, **P4**, is `lastChange(P2(1):color)`. Figure 5.10 shows the corresponding call stack and source code at **P4**. **P4** is the fault, because the value of green slider is used as the value of blue slider and it is the reason of the wrong color.

```
----- Collected Data
P1:this <- ObjectId:1550, Class:SunGraphics2D
P2:.owner  <- ObjectId:1550, Class:SunGraphics2D
P2(1):this <- ObjectId:1536, Class:PencilPaint
P3:.owner  <- ObjectId:1536, Class:PencilPaint

----- Call Stack
    PaintObject.java:setColor():10
    PaintObjectConstructor.java:mousePressed():60
    Component.java:processMouseMotionEvent():6261
    JComponent.java:processMouseMotionEvent():3283
    ...
    EventDispatchThread.java:run():122

---- Source Code
9   public class PaintObjectConstructor implements MouseListener, MouseMotionListener {
...
13      private PaintObject temporaryObject;
...
52      public void mousePressed(MouseEvent e) {
...
60          temporaryObject.setColor(color);
...
66      }
...
100 }
```

Figure 5.9: Data collected for identifying P3, the thread call stack and corresponding source code.

```
----- Call Stack
     PaintObjectConstructor.java:setColor():26
     PaintWindow.java:stateChanged():26
     JSlider.java:fireStateChanged():420
     JSlider.java:stateChanged():337
     ...
     EventDispatchThread.java:run():122

---- Source Code
10  public class PaintWindow extends JFrame implements PaintObjectConstructorListener {
...
23  private PaintObjectConstructor objectConstructor;
...
25      public void stateChanged(ChangeEvent changeEvent) {
26        objectConstructor.setColor(new Color(rSlider.getValue(),
27                                gSlider.getValue(), gSlider.getValue()));
28       repaint();
29      }
...
165 }
```

Figure 5.10: The thread call stack and corresponding source code of P4.

Figure 5.11 is a screenshot of the execution trace viewer in the prototype debugger, taken after applying the four querypoints described above. The smaller circles are those points which are inspected but they have not satisfied all the constraints. The bigger circles demonstrate points **P1** to **P4**. Due to non-determinism, none of the points **P2**, **P3** and **P4** are recognized before identifying **P1**. Therefore, a developer cannot inspect program state by pausing at those points. Instead, a developer uses this interface and asks the debugger to collect needed data in the next re-execution.

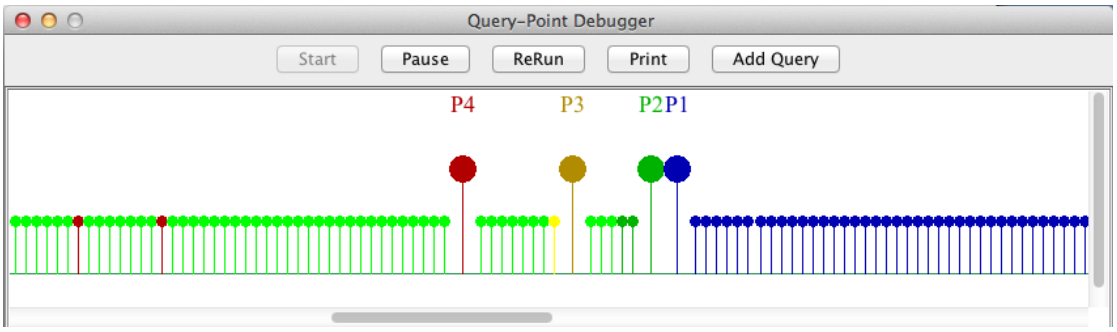


Figure 5.11: The screenshot of the execution trace viewer in a the prototype querypoint debugger applied to the application shown in Figure 5.2.

5.3 Related Work

Several other query languages and mechanism have been developed for monitoring an execution and locating execution points with particular featurs. Among them we can mention PQL [Martin2005] and tracematch [Allan2005]. One main difference of our language with these languages is that a query can be defined by referring to the result of previous queries and this feature is essential in implenting a querypoint debugger.

Don't despise empiric truth. Lots of things work in practice for which the laboratory has never found proof.
— Martin H. Fischer

6 Debugging by lastChange

In this chapter we explain the details of our *lastChange* querypoint implementation for JavaScript, which is a weakly-typed language. Moreover, we show how this querypoint is integrated into Firebug, a traditional JavaScript debugger. At the end, we report the results of the user-study that we conducted in order to evaluate the effect of *lastChange* querypoint on the time and effort needed in debugging.

6.1 Introductory Example

We illustrate the *lastChange* functionality by a simple example. The example demonstrates a buggy JavaScript code in a HTML page (Figure 6.1). The page contains a button (line 40) showing the value of `myObject.myProperty`. When the user clicks on the button, the `onClick` function (line 13) is called. This function increases the value of `myObject.myProperty` by one (line 15) and calls `updateButton` function which updates the button's text to the new value (line 22). Once the page is loaded for the first time the button shows `1` as the initial value of `myObject.myProperty`. In practice when the user clicks on the button, `0` appears instead of 2: there is a bug.

Two other functions are called in `onClick()`, `foo()` and `bar()`. As developers we often encounter function calls which seem peripheral to our current concern; they may have been added by another developer, or we may have forgotten their exact properties or those properties may have changed, and so on. The difference between what we expect these functions to do, e.g. nothing interesting, and what they do in practice may cause bugs.

By browsing through the code or other means[Barton2010], the developer determines that the value displayed on the button is set at line 22. Since the displayed value is incorrect we know the bug occurred before we hit this line. To start debugging, the developer sets a breakpoint on line 22. Once the button is clicked, the execution is paused at line 22. Figure 6.3a shows the Firebug debugger while the execution is paused. Firebug has several panels (e.g., HTML, CSS, Script, DOM, etc.) that each demonstrate one aspect of the Web page. The

```
<html>
    <head>
        <title> lastChange Introductory Example
        </title>
        <script type="text/javascript">
            myObject = {
                        myProperty : 1
            };
            myCondition = {
                        value : 1
            };
            var oldValue;
            function onClick(){
                foo();
                myObject.myProperty++;
                bar();
                    //...
                updateButton();
            }
            function updateButton(){
                var myParagraph = document.getElementById("myButton");
                myButton.innerHTML = myObject.myProperty;
            }
            function foo(){
                myCondition.value = oldValue;
            }
            function bar(){
                if (!myCondition.value)
                    myObject.myProperty = 0;
            }
        </script>
    </head>
    <body id="myBody">
        <p>
            <ol>
                <li> Click on the button.
                </li>
            </ol>
        </P>
        <button id="myButton" onclick="onClick()">
        1
        </button>
    </body>
</html>
```

Figure 6.1: A Web page containing JavaScript code.

Script panel contains the list of all loaded source files and regular debugging facilities such as setting breakpoints and stepping. To the right of the script panel, the Watch panel shows the program state where the developer can examine object and variable values. In our case, the `myObject.myProperty` value at the paused point is `0`. We expected this value to be 2.

To apply backward search strategy for locating defects, the developer first needs to know the origin of the wrong value. To achieve this goal using breakpoints, the developer should search code to find all possible places that `myObject.myProperty` might get a new value and set breakpoint at these locations. However, an object and property can be accessed and changed through different names and methods. There is no simple way to identify these aliases or even their total number. The developer can make a good guess and set breakpoints on lines where the property seems to be changed. Then they re-execute the program and examine the state looking for values that may lead to the incorrect value observed at line 22. All this work must be repeated if a new alias is discovered or if some information related to the buggy result was missed while stopped on one of the breakpoints.

In contrast, we have added a high-level function in the debugger, *lastChange*, which provides the answer without tedious manual effort from the developer. By right clicking on `myObject.myProperty` in the Watch panel, the developer can run *lastChange* command (Figure 6.3a). The debugger re-executes the program and halts again at the breakpoint on line 22. However, it shows a new panel, called QP, centered on the source at line 29 (Figure 6.3b), the point of *lastChange*. To the right, the TraceData panel shows values of properties of the program state when it passed through line 29. These two panels resemble the Script and Watch panels, but they show data collected by the debugger at one execution point which is now past: these are *traces* or *logs* of information collected during the re-execution.

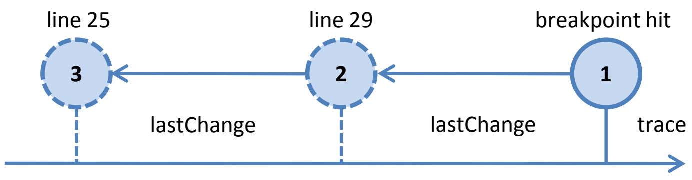


Figure 6.2: The examined points before locating the defect. The arrow represents the logical forward progress of the program. Three actual executions are superimposed on this arrow. All three stop at the reproduction point indicated by circle 1. After the first execution, the developer asks for lastChange as described in section 6.1, yielding information indicated by circle 2. After the second execution, another lastChange query causes a third execution, yielding information indicated by circle 3.

Looking at line 29, it seems that something is wrong with `myCondition.value` which causes line execution. The developer examines `myCondition.value` and it is `undefined`. The next step is to know when this property got this value. To do so, the developer runs the *lastChange* command on `myCondition.value` at this point. The debugger re-executes the program and breaks again on line 22, analyzes its queries and shows the developer line 25-the place `oldValue` is assigned to `myCondition.value`. If the developer asks for *lastChange* on `oldValue`, the debugger can notify the developer that this variable is never assigned a value. Now it is clear that the bug occurs because `oldValue` is `undefined` once the execution reaches line 25 (Figure 6.3c).

As demonstrated in Figure 6.2, the developer has examined three points of execution. The first point was the breakpoint set by the developer. We call this special breakpoint the *reproduction point*. The second and third points preceded the reproduction point in execution sequence. All three points-the history of the search for the defect-are available through the debugger's interface. On the top of the left panel in Figure 6.3c there is an opened list which shows all three examined points. The first one is the breakpoint on line 22, the second one is the point which is when `myObject.myProperty` changed before reaching the breakpoint and finally the last one is the point of execution in which `myCondition.value` gets the `undefined` value. Moreover, the source lines related to these points are marked with red **Q** icons.

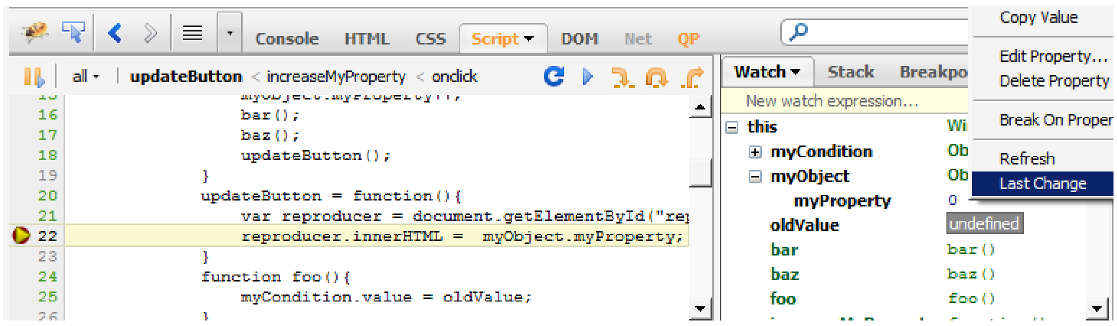
Notice that in our example, *lastChange* combines some aspects of breakpoint-based and of log-based debugging. Like breakpoint-based debugging, the developer re-executes a live runtime without changing the source and without a special execution environment beyond the debugger. The state of the program memory and the call stack are available at each lastChange point. Like log-based debugging, the program state and the call stack are recorded during program execution. We can't halt the program at *lastChange* because we don't know which point is the last one until we return to the original breakpoint. In section 5 we discuss cases where it is possible to pause at lines of *lastChange*.

6.2 lastChange Algorithm

The *lastChange* algorithm is based on program re-execution of a program halted on a breakpoint. The algorithm starts when the developer examines the program state at a breakpoint hit and asks for the *lastChange* of a value. The breakpoint hit becomes the *reproduction point*. Debugger sets hooks (a callback function dependent upon the underlying runtime) on all instructions that might be the result of *lastChange* query. Then the debugger re-executes the program and every time a hook hits it checks for a *change event*. In the case of a change, it stores part of the program state values. Once the execution reaches the reproduction point, it analyzes the collected data and shows the result. The program state at the execution point of the last change event is the *lastChange*.

As we described in the preceding section, a *lastChange* query can be performed on the result of another *lastChange* query. If we name the reproduction point *R*, we can write the first *lastChange* in the introductory example in this form: *lastChange(R, myObject.myProperty)*. It means that this query is defined at *R*. If we name the result of this query *L*, we can write the second *lastChange* in this form: *lastChange(L, myCondition.value)*. In this way, a sequence of *lastChange* queries with any length can be defined.

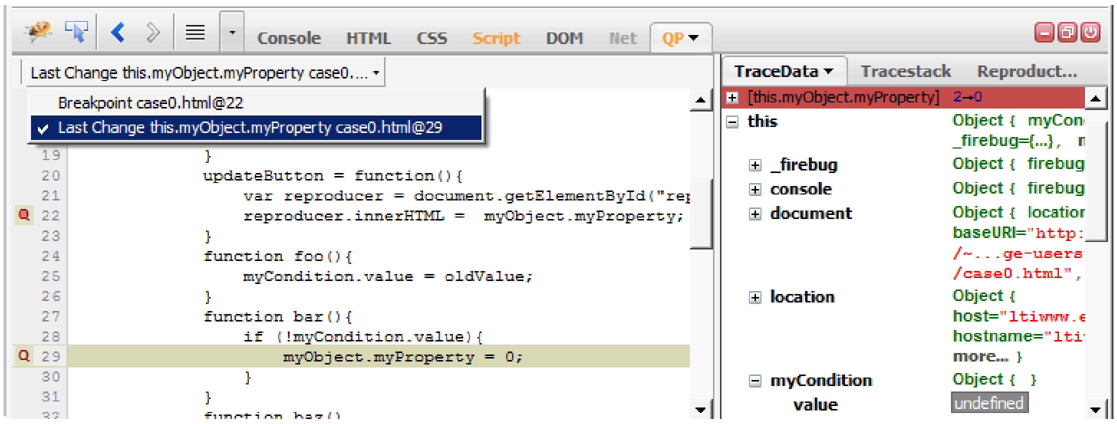
lastChange can be called on object property, on a variable value, or on the results of a



```
16              bar();
17              baz();
18              updateButton();
19          }
20          updateButton = function(){
21              var reproducer = document.getElementById("rep
22              reproducer.innerHTML =  myObject.myProperty;
23          }
24          function foo(){
25              myCondition.value = oldValue;
```

```
New watch expression...
this                Wi
  myCondition       Ob
  myObject          Ob
    myProperty      0
  oldValue          undefined
  bar               bar()
  baz               baz()
  foo               foo()
```

(a) A screen shot of the Firebug debugger while running the example code from Fig. 6.1. The Script panel is selected; it gives access to all loaded source files and allows breakpoints to be set on lines. In this figure, the execution is paused at line 22 by a regular breakpoint. The Watch panel on the right shows the program state at the paused point. Developer can query *lastChange* on `myObject.myProperty` by right-clicking on the value of `myProperty`.

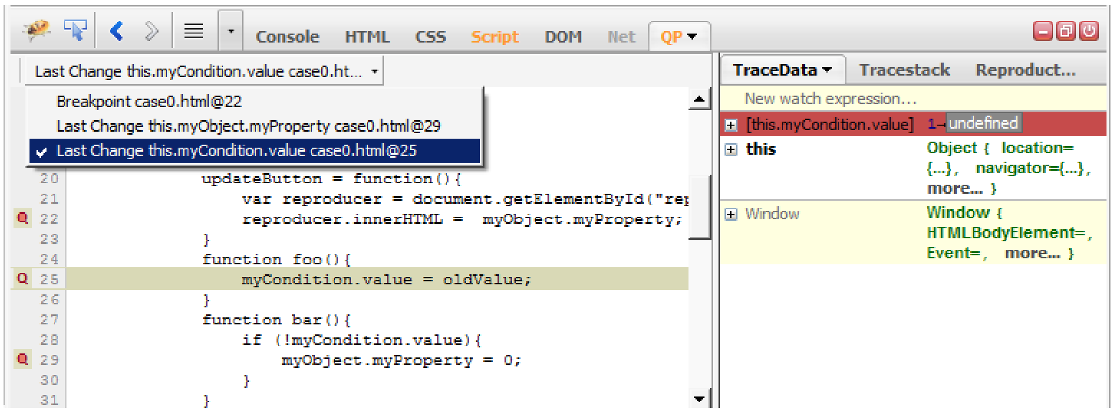


```
Last Change this.myObject.myProperty case0....
    Breakpoint case0.html@22
  ✓ Last Change this.myObject.myProperty case0.html@29
```

```
19          }
20          updateButton = function(){
21              var reproducer = document.getElementById("rep
22              reproducer.innerHTML =  myObject.myProperty;
23          }
24          function foo(){
25              myCondition.value = oldValue;
26          }
27          function bar(){
28              if (!myCondition.value){
29                  myObject.myProperty = 0;
30              }
31          }
```

```
[this.myObject.myProperty]  2→0
this                Object { myCon
                    _firebug={...},  n
  _firebug          Object { firebug
  console           Object { firebug
  document          Object { locatior
                    baseURI="http:
                    /~...ge-users
                    /case0.html",
  location          Object {
                    host="ltiwww.e
                    hostname="lti
                    more... }
  myCondition       Object {  }
    value           undefined
```

(b) The result of *lastChange* query for `myObject.myProperty`. The left panel, QP, shows the source code at the point of *lastChange*; The right panel, TraceData, shows the collected data at the point.



```
Last Change this.myCondition.value case0.ht...
    Breakpoint case0.html@22
    Last Change this.myObject.myProperty case0.html@29
  ✓ Last Change this.myCondition.value case0.html@25
```

```
20          updateButton = function(){
21              var reproducer = document.getElementById("rep
22              reproducer.innerHTML =  myObject.myProperty;
23          }
24          function foo(){
25              myCondition.value = oldValue;
26          }
27          function bar(){
28              if (!myCondition.value){
29                  myObject.myProperty = 0;
30              }
31          }
```

```
New watch expression...
[this.myCondition.value]  1→undefined
this                Object { location=
                    {...},  navigator={...},
                    more... }
Window              Window {
                    HTMLBodyElement=,
                    Event=,  more... }
```

(c) The result of *lastChange* query for `myCondition.value`. To evaluate an expression (e.g., oldValue) at this point, developer can enter the expression in the watch box and after re-execution the result is available. The opened list on the top of the left panel shows the visited execution points. Clicking on each point in the list shows the corresponding code and data.

Figure 6.3: The stages of locating the defect using the *lastChange* feature.

lastChange. Moroever, common data structures such as arrays and hashmaps are also supported as special cases of *lastChange* on object property. We explain each case in the following subsections.

6.2.1 *lastChange* on Object Property

To simplify the algorithm explanation and defer technical details, we define two basic operations and later we explain the details of these two operations. The first operation is `objectId()`: given a JavaScript object it returns an integer as its identifier. This identifier is unique to the object during one execution. By using an object id instead of an object reference we allow the garbage collector to reclaim the space for dead objects just as it would in the absence of the debugger. The second operation is `setPropertyChangeHook()`: given a function and a string, the function is called whenever a property changes and its name matches the string. For example, if the string is `foo`, changes to `bar.foo` or `baz.foo` would call the function. The callback function receives a reference to the owner of `foo`.

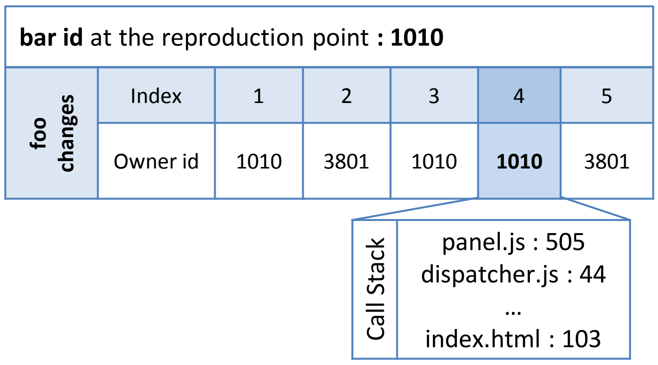
To see how these functions work, suppose the developer asks for the last change of `bar.foo` at the reproduction point in a program. The debugger calls `setPropertyChangeHook()` with `foo` as the property name and re-executes the program. Whenever `foo` changes and the callback function is to be called, debugger first calls `objectId()` on the `foo` owner object. Then it stores this owner id, the stack frame locations, and other state values in scope at the call point. Then the callback returns to continue the execution. Thus the query is not a breakpoint in the sense of pausing for user interaction, but breakpoint technology can be used to implement the query. Whenever the execution reaches the reproduction point the debugger looks at the history of `foo` changes and finds the last `foo` change with the same object id as `bar` id at the reproduction point. Figure 6.4 shows the list of property `foo` change events in a hypothetical execution. `bar` id at the reproduction point is 1010, so the last change of `bar.foo` is the fourth column.

6.2.2 *lastChange* on Variable

In JavaScript, every frame has a scope chain and every available variable in the frame comes from one of the scopes in the frame's scope chain. Once the developer asks for the last change of a variable with name `foo` at the reproduction point, the debugger first determines the variable's scope as follows: it iterates over the scopes in the scope chain and the first scope which has a variable with the same name is the variable's scope. There are five different scope types: global, local, closure, `with` and `catch`. We explain these cases in two groups.

global and `with` scopes

Global scope is the most outer scope in the scope chain and it is also referred to as the global object (the `window` object in Web pages). This scope is a regular JavaScript object and therefore



bar id at the reproduction point **: 1010**

foo changes					
Index	1	2	3	4	5
Owner id	1010	3801	1010	**1010**	3801

Call Stack
panel.js : 505
dispatcher.js : 44
...
index.html : 103

Figure 6.4: A hypothetical list of change events for a property `foo`. Each change event adds a column with the id of the object changed, the call stack, and some program state such as local variable values. At the reproduction point we determine which id corresponds to object `bar` and read out column 4, the last change of `bar.foo`. Column 3 is also a change of the object we want to study, id 1010, but it is not the last change; Column 5 is also a change of a property `foo` but it is not for the object we are interested in.

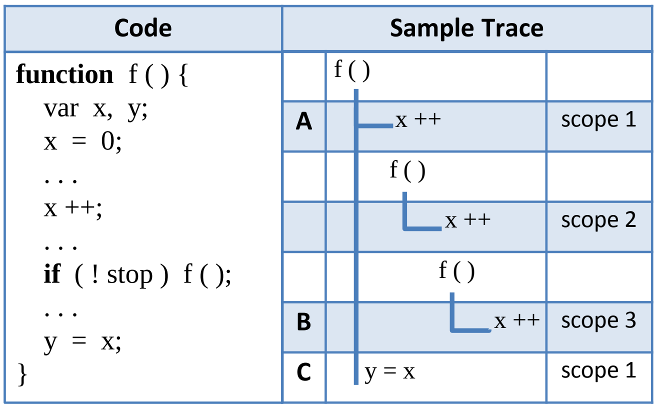
every global variable is a property of global object. Similarly, `with` scopes are also regular JavaScript objects. A `with` scope is created by a `with()` block with an object as the parameter. Every property of this parameter object is available inside the block as a variable. *lastChange* treats the case where variable's scope is global or `with`, like it does on an object property.

local, closure and `catch` scopes

Local scope refers to the most nested scope in the scope chain which contains the local variables. Closure scope refers to the scope which is created for a nested function and contains variables defined in the outer block. Catch scope is the scope created in the catch block of try-catch statements and contains the exception variable. These scopes are not necessarily regular JavaScript objects. Therefore, to track changes to a variable in these scopes we employ a different approach.

Having the scope chain and the source code, we can map every scope to a code block, enclosed the executing code. In JavaScript, a code block can be identified by the file url and the block's first instruction program counter. Given this information, the debugger is able to recognize the code block in loaded scripts or once it is loaded. Similar to *lastChange* on object property, we define two basic operations: `scopeId()` – which returns an integer given a scope – and `setVariableChangeHook()` – which calls its first parameter, a callback function, when a variable in its second parameter, a code block, matches the name given in the third parameter.

Figure 6.5 illustrates how the `scopeId()` operation separates instances of a variable in different scopes having the same name. If we ask for the last change on `x` at point labeled `C` in scope with id 1, we want the change at line `A` in scope 1, not the change at the line marked `B`, where a variable named `x` in scope 3 is changed.



```
function f ( ) {
   var x, y;
   x = 0;
   . . .
   x ++;
   . . .
   if ( ! stop ) f ( );
   . . .
   y = x;
}
```

Figure 6.5: A recursive call trace illustrating the scope id. The lines in the bottom half of the diagram simulate a trace of the change events for the variable `x` as the function `f()` calls itself. Each call creates a scope; eventually when the variable `stop` is changed by an external process we return from the recursion. The lines marked A, B, and C are discussed in the text.

If the developer asks for the last change of variable `foo` at the reproduction point in a program, debugger calls `setVari- ableChangeHook()` with the variable's defining block and name as parameters and re-executes the program. Whenever `foo` changes and the callback function is to be called, debugger first calls `scopeId()` on the variable's scope. Then it stores this scope id, the stack frame locations, and other state values in scope at the call point. Whenever the execution reaches the reproduction point the debugger looks at the history of `foo` changes and finds the last `foo` change with the same scope id as the variable's scope id at the reproduction point (Figure 6.6).

foo scope id at the reproduction point : **55**						
foo changes	Index	1	2	3	4	5
	Scope id	50	**55**	36	50	100

Figure 6.6: A hypothetical list of variable `foo` change events. Each column of the list indicates a change event; for each change event the scope id returns by `scopeId()` is recorded along with call stack and program state information. The last column having the scope id `foo` at the reproduction point indicates the last change.

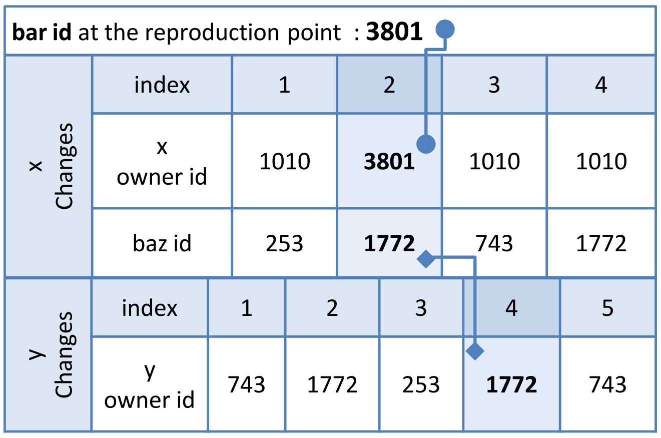
6.2.3 *lastChange* on *lastChange*

The *lastChange* algorithm records changes by id (either object or scope id), then reads out the last change when we arrive at the reproduction point and discover the id of the requested value. When we perform *lastChange* based on a previous *lastChange*, the query algorithm must retain additional information. Consider the following example:

```
point A : the reproduction point
point B : lastChange(A, bar.x)
point C : lastChange(B, baz.y)
```

where point A is a breakpoint, point B is the last change of the object property `bar.x` at point A, and point C is the last change of the object property `baz.y` at point B. The object referenced by `baz` changes upon re-execution. Therefore when the developer asks for the last change of `baz.y`, we need to track objects named `baz` at changes of `bar.x` and changes to objects named `y`. Then, at the reproduction point, we need to work out which `baz` the developer wanted, then select the last change of that `baz.y`. Figure 6.7 illustrates the extra row of data (tracking of `baz` objects at `x` change events) and how the id values allow the last change of `baz.y` to be worked out.

In the general case we perform dependency analysis as outlined in Figure 6.8 to create the list of additional data (object id or scope id) to be collected at a change event. The process can be repeated to cascade *lastChange* arbitrarily deep.



bar id at the reproduction point : **3801**

x Changes					
index	1	2	3	4	
x owner id	1010	**3801**	1010	1010	
baz id	253	**1772**	743	1772	

y Changes					
index	1	2	3	4	5
y owner id	743	1772	253	**1772**	743

Figure 6.7: The list of change events stored for locating point B, the *lastChange* of `bar.x` at the reproduction point, and point C, the *lastChange* of `baz.y` at point B.

6.3 JavaScript implementation

To verify the *lastChange* algorithm we implemented[1] it in an extension to the Firebug JavaScript debugger[2]. Firebug itself is an extension of the Firefox browser. The Firefox JavaScript engine provides a JavaScript debugging interface and *Querypoint* is developed over this interface. Our prototype implements the four primitive operations in JavaScript using techniques which are

[1]http://code.google.com/p/querypoint-debugging
[2]http://getfirebug.com

```
for $q$ in $lastChange$ queries do
  for $p$ is defined at the $q$ result do
    if $p$ is a $lastChange$ on object property then
      the property owner id must be stored at $q$ change
      events.
    else if $p$ is a $lastChange$ on variable then
      the variable scope id must be stored at $q$ change
      events.
    end if
  end for
end for
```

Figure 6.8: *lastChange* queries dependency analysis.

cumbersome and comparatively slow to execute. However the JavaScript prototype is easy to explore, change, and share with others for feedback. A professionally useful debugger would implement these primitive operations within the JavaScript engine.

6.3.1 `objectId()` Operation

`objectId(obj)` first checks the argument `obj` for a property `_objectId`. It returns the value if this property is already defined, otherwise it generates a new id and sets this property. The value of the id is simply an integer incremented for each new `_objectId` needed.

6.3.2 `setPropertyChangeHook()` Operation

The Firefox JavaScript engine supports watching property changes in an object. Every object has a function `watch(propName, callback)` which receives two parameters, a property name and a function. Whenever the property with the given name changes, the `callback` function is called. The hook set by this function remains enabled even if the property is deleted and defined again.

For our purposes, the `watch()` function only covers the case of global object properties. At the beginning of execution, no object excepting the global object and its predefined properties is available. For our *lastChange* prototype, we created a version of `setPropertyChangeHook()`. The basic strategy is to get a reference to the object just after its creation, then use `watch()` function to monitor property changes in the object. Setting a flag[3] into the Firefox JavaScript engine, we can get the file URL and the line number for each object creation (e.g., myFile.js, line 24). We set a breakpoint on this line and parse the source code to determine which object was created.

The only data we have is the object creation location including the file url and the line number

[3]DISABLE_OBJECT_TRACE defined in jsdIDebuggerService.idl

and the goal is to get a reference to this object. Although in most regular cases we have only one statement and one object creation, there are cases where more objects are created in the line. There is no simple way to recognize the interesting object among these new objects. So instead of one object, we monitor all new objects created in the line.

An object might be created by one of these statements: object literal(`{...}`), *new* operator (`new constructor()`) or function definition statement (`function()`). By parsing the source code we can recognize the statements that create an object. The next step is getting a reference to the new object.

The new object can be assigned to an object property or a variable by an assignment (=). In these cases we keep the assignee statement at the left side. The idea is that we create a list of assignee statements that the new objects are assigned to. We set a hook on the creation line. Once the hook hits, we evaluate the assignee statements. Then we do stepping(step-over) and after each step we evaluate the statements. Every statement which has a new value, we consider the new value (if it is an object) as a new object. For example in Figure 6.9(a), if the creation line number is 20, we have only one statement which creates a new object and it is assigned to `x.y`.

The new object can also be set as the property of a parent object (or array) inside an object (or array) literal. This case is also treated similar to previous case. The only difference is that the full path of property from the root parent in the local scope must be considered as the assignee statement. For example in Figure 6.9(b), if the creation line is 20, we keep `parent.child`. In this case stepping must be continued until the end of the object literal (line 22) for getting a reference to the created new object.

In cases where the new object is passed as an argument to a function (Figure 6.9(c)), we use step-in instead of step-over. To get a reference to the new object it is enough to evaluate the corresponding argument inside the function. The other cases where the program does not keep a reference to the created object (e.g., when a function object is just called after its creation), are not in our interest.

Although this approach is successful in many ordinary cases, we can imagine cases where a more comprehensive analysis needed for the correct behaviour. Consider the case where the new object is assigned to an expression like `a[++i]`. Obviously, evaluating this statement doesn't return a reference to the new object. Our prototype implementation does not handle these kinds of unusual cases yet.

Throughout this section we implicitly assumed that the object will be created at the same location in the next execution. If the assumption is not true, once the execution reaches the reproduction point, it reveals that the object has been created in a different location. This time, prototype re-executes the program considering both locations as possible object creation locations.

```
(a) Case 1:
  x.y = new MyClass();

(b) Case 2:
  parent = {
    child : {
      x : 5
  }}

(c) Case 3:
  myFunction({myProperty:5});
```

Figure 6.9: Examples of different cases in getting references to created JavaScript objects.

6.3.3 `scopeId()` Operation

The prototype sets a breakpoint at the beginning of all code blocks needing a scope id. The scope id is kept as a variable with name `_scopeId` in the scope. Whenever the hook is hit, meaning a new scope is created, `_scopeId` is set by calling JavaScript's dynamic compilation function `eval()`. For example, executing `eval("var _scopeId = 10")` creates a variable with name `_scopeId` and value `10` in the scope of the `eval()` call, which is our interesting scope. `scopeId()` operation returns the value of `_scopeId` in the scope.

6.3.4 `setVariableChangeHook()` Operation

Variables defined in local, closure, and catch scopes are only changed in their scope; that includes the defining scope and scopes nested whithin them. For example in Figure 6.10, variable `foo` in line 11 can only be changed in lines 10 to 23. Therefore, after locating the function in which the variable is defined, it is enough to parse the code inside the function block and set a hook on all lines where the variable is assigned a new value. We also set a hook at the first line of the function which is corresponding to the line where the variable is defined. In Figure 6.10, if the execution is paused at line 17 and the last change of `foo` is queried, two hooks on lines 16 and 17 will be set, but if the execution is paused at line 20 , four hooks on lines 11, 12, 14, 20 will be set. These two cases are different: `foo` in `childOne` is a local variable but in `childTwo` it is a closure variable.

6.3.5 Re-Execution, Reproduction Point and Data Collection

Querypoint needs a test case to reproduce the execution and conditions to correctly recognize the reproduction point. Although both elements can be directly provided by developer, *Querypoint* is also able to automatically create them from the first execution.

To replay execution, *Querypoint* keeps track of breakpoint hits and single steps. For example,

```
function main() {
    var foo;
    foo = ...;
    function parent() {
        foo = ...;
        function childOne() {
            var foo;
            foo = ...;
        }
        function childTwo() {
            foo = ...;
        }
    }
}
```

Figure 6.10: Sample JavaScript code demonstrating local and closure variables.

if the developer queries *lastChange* at the third hit of breakpoint *b*, in re-execution, the third hit is recognized as the reproduction point.

The *Querypoint* prototype supports two mechanism for automatic re-execution: callstack-reproduction and record-replay. In callstack reproduction the function from the earliest frame of the call stack is called with the same parameters. The idea behind this mechanism is that many bugs in web pages can be reproduced by re-firing an event like clicking on a button. The record-replay execution uses two phases. In the record phase, it stores the initial page url and the events and parameters corresponding to user actions. In the replay phase, it opens the same url and simulates events as if they were user actions. The callstack reproduction mechanism provides shorter re-execution cycles while the record-replay is more accurate about the initial state.

In addition to the data collected at every change event for identifying the *lastChange* result, *Querypoint* partially stores values in program state. There is a trade-off between the amount of data collected at every change event and the number of re-executions. If developer asks for some values which have not been stored, *Querypoint* re-executes and collects the requested data.

6.4 Reproducible Non-Deterministic Execution

We claim that the only prerequisite for *lastChange* is bug-reproducibility. A bug is *reproducible* for a developer when the developer can start from a given initial state, operate on the program with a list of actions, and reproduce the symptoms of the bug. The details of the execution can change each time we re-execute the buggy program, but the buggy result is the same. All modern debuggers in wide use that we know about rely on reproducible but non-deterministic execution for simple practical reasons: developers must reproduce a bug to study it and modern execution environments are not deterministic. By relying on reproducibility but not requiring deterministic execution, *lastChange* works on the same range of cases.

6.4.1 *lastChange* Result Consistency

Each time we re-execute a non-deterministic program, the details of execution instruction order may change. For example, if we record the source code lines every time a conventional watchpoint hits, the record may differ each time we re-execute. But *lastChange* does not compare values across executions. Rather it analyzes all of the change events in a single execution. Neither the data gathering nor the analysis require deterministic execution.

Since we don't need to compare values across executions to implement *lastChange*, we can get more information if we do compare: different points of last change on different executions will signal that the execution is not deterministic. Note the converse is not true, many non-deterministic programs will give identical results for *lastChange*. Consider the example in Figure 6.11. Assume that at the first execution, the developer sees that the `a` value is `true` and asks for the *lastChange* of `a` at the breakpoint. In the re-execution, `a` is `false`, and `b` is `true` at the reproduction point. It means that `lastChange` result shows line 10, which is a correct answer for this execution but not for the previous one. When user asks for the *lastChange* on variable `a`, debugger stores `a` value and compares it to the `a` value in the next executions. So if this value is different, debugger informs the developer that the *lastChange* query is made on a different value.

```
10  a = b = false;
11  if (random()) {a = true;} else {b = true;}
12  if (a || b) {bug(); /* breakpoint */}
```

Figure 6.11: *lastChange* on non-deterministic values.

6.4.2 Combination of *lastChange* and Breakpoint Debugging

Using *lastChange* a developer can work backwards on the flow of data, but sometimes bugs are more obvious when we watch control flow forwards. Consider for example,

```
44  vector.r = Math.floor(Math.random()*5)*6;
45  if (vector.r !== 0 && vector.r < 30);
46      return vector;
```

where *lastChange* shows us that `vector.r` is zero at line 44. In our user studies, developers wanted to single step forward from line 44. If they could do this, they may be surprised to see line 46 execute even if `vector.r` is zero, directing their attention to line 45 where they can discover the errant semicolon. However, a *lastChange* is just a query result, not a breakpoint. Under what conditions can we cause the debugger to stop at the *lastChange* point?

In the case that the execution is deterministic, the *last- Change* event index can be used as an index for a conditional breakpoint [Boothe2000, Maruyama2003]. Moreover, this works in a non-deterministic execution if non-determinism has no effect on the event index. The debugger can easily set this conditional breakpoint and reexecute the program. If the stream of change events differs in this re-execution the developer can be warned that the conditional breakpoint may not be the *lastChange* point. However, if the stream of events is the same, we do not know that the conditional breakpoint matches the last change, this fact can only be verified at the reproduction point. These theoretical concerns are not likely to cause significant practical problems: if the value at the conditional breakpoint is not suspicious, then the developer will know to simply return to *lastChange* or move on to another tactic to find the bug.

6.5 User Study

We supplied four experienced Javascript developers with our prototype in an extended Firebug debugger[4]. Following a tutorial and a practice case, we observed as they applied both conventional breakpoint and *lastChange* on two small programs we provided (Our prototype did not have a user interface to support Sec. 6.4.2). The first program, Shapes, calculates the area and perimeter values for a list of shapes. The bug happens when one of the calculated numbers is zero. The second program, Moving Circle, randomly scales and moves a circle in the page. The bug happens once the circle becomes invisible after an exception occurs. This case represents a reproducible non-deterministic execution. The developers were asked to locate the defects that caused these bugs. All four developers successfully applied *lastChange* to the test programs and understood how it could help debugging.

To find the defect location with breakpoints, all four users took more steps and more time (Figure 6.12). Two users scrolled through the source, another searched the text, a third set a lot of breakpoints to understand the control flow. Based on our own experience we expect these strategies represent the kinds of approaches developers have available. These operations are time consuming and tedious. In contrast, all four users found the defect location with just two *lastChange* operations. We recognize that these programs were designed to highlight lastChange and many kinds of debugging issues have been hidden by the design of our tests. Nevertheless our results show that, when a defect relates to incorrect values and a developer recognizes this, then the operational mechanics of lastChange lead to the defect much more quickly than breakpoints.

Our observation and discussions with users also brought out several important issues and im-

[4]The user study description and source files can be found in appendix A.

provements for our user interface. Perhaps the most important and challenging improvement would be better integration with breakpoint debugging. Our implementation put the results of *lastChange* in a similar but different view from breakpoint debugging. This focuses attention on value changes, but it makes studying control flow more difficult: we don't support single stepping from a *lastChange* result in our user interface. In our next iteration we plan to merge the query and breakpoint results and support pausing as described in Sec. 6.4.2.

Developers / **Programs**	**DEV1**	**DEV2**	**DEV3**	**DEV4**
Shapes	**3 (85 s)**	13 (182 s)	39 (318 s)	**3 (80 s)**
Moving Circle	9 (215 s)	**4 (40 s)**	**3 (195 s)**	12 (234 s)

Figure 6.12: The number of steps (and time in seconds) required before locating a defect, for each test subject and test program. Cells with a white background report values with conventional debugging; Cells with a colored background use *lastChange*.

6.6 Discussion

We have presented the *lastChange* algorithm and described our prototype implementation. Our goal is practical improvements in debugging. To achieve our goal we need practical JavaScript engines to add new debug primitives so that developers in the field can use our new technique. This paper is one step to convince implementers to enable *lastChange*. Thus we summarize here our arguments that *lastChange* should be supported.

6.6.1 Developers need an operation like *lastChange*

Since ultimately programs are just transformation of state values, debugging is ultimately backtracking to find defects in program state change[Weiser1979]. When a developer halts a program on a breakpoint they compare the call stack and program state to their model of the program. If any value seems to be incorrect, they need to figure out what operation causes the incorrect value. Here *lastChange* takes over and addresses a key part of the debugging process.

6.6.2 Developers can learn to use *lastChange*

While our user study was small, our prototype demonstrates that *lastChange* can be easily activated by operations on the graphical representation of erroneous data in a debugger and the results can be interpreted by users. While *lastChange* is not a breakpoint, all of the assumptions developers already have for breakpoints hold for *lastChange*. In particular the re-execution is just the same operation developers use to debug with breakpoints.

6.6.3 Practical implementations are feasible

We have described our prototype all-JavaScript implementation in Sec. 4. It is adequate for exploring the ideas and may even be usable in production. However significant improvments can be made. A fully usable implementation would require access to the object id at the point of object creation and `setPropertyChangeHook()`. Many object-oriented runtimes provide object identifiers and provide access to object creation directly or by bytecode instrumentation.

6.6.4 In most cases *lastChange* will be much faster than current alternatives

Recall that we insert additional code through debugger callbacks, then re-execute the program. The additional code we insert is proportional (in our JavaScript algorithm) to 1) the number of places a property or variable with a given name is changed, 2) the number of places objects are created. The overhead for each execution at a change event depends upon the amount of data we store for each change event; the overhead for object creation could be small since we only need to determine if the object is one we need to watch.

For comparison consider today's practical alternative: developers setting breakpoints. For the vast majority of programs, a developer will take much more time to set one breakpoint than *lastChange* would add. Moreover, typically the developer may not guess the point of last change. They must then ponder another breakpoint and re-execute.

We could also compare to solutions based on logging or tracing. Manual logging has very high overhead: the developer must add code, debug that added code, then analyze the log (To be fair, the log can become a permanent debugging aid). Automatic logging causes about one or two orders of magnitude slow down as well as requiring a completely different set of development tools.

6.6.5 The worst cases are not more common or more painful than alternatives

Every time through the loop we incur the call back overhead; if the loop itself has relatively little code the overhead could be very large; if the loop computation is a significant fraction of the full program, the slowdown would be enormous. Other techniques also struggle with this case: breakpoints in highly repeating loops are not feasible and logging becomes unwieldy. Developers face this issue with any debugger today: occasionally a debugger causes too much overhead to be useful for debugging.

6.6.6 Generalizations

Our *lastChange* algorithm can be viewed as a particular interface to a general facility. The general facility replays execution, queries the runtime at points of interest during execution, and analyzes the result at the reproduction point. We have selected one kind of query and analysis that can be easily integrated in existing debuggers and explained to developers, providing automation of the problem of finding the point of last change. We believe other kinds of queries and analysis can be invented and integrated to automate other aspects of debugging.

I didn’t realize it was possible people could be so anonymous. There was no recognition of a person as an individual. I didn’t like it. It’s not a good learning environment.

— Ashley Mason

7 Naming Anonymous JavaScript Functions

The unique and important role of JavaScript in web programming is undeniable. Along with the wave of “Web 2.0," JavaScript has become the inevitable part of almost every modern web site[1]. It is very likely that JavaScript keeps this crucial role for the next few years or even the next decade. Due to the growth of demands for more comprehensive user interfaces, the size and the complexity of web applications are increasing. Moreover, JavaScript is also becoming a general-purpose computing platform [Richards2010] for office applications [JS-MSOffice, JS-OpenOffice], browsers [FirefoxAddons, GoogleChromeExtensions], program development environments [Ingalls2008], and even server-side applications [CommonJS, ServerSideJS].

To cope with these large and sophisticated systems, JavaScript developers turn to development tools. One prime example is a runtime debugger: The developer can halt a running program and examine the program state and execution call stack. All of these tools need to express program artifacts in a compact way that the developer can understand. For example, the debugger must present the execution call stack, so the developer can understand which functions are currently active. Obviously, a particularly good compact representation would be a name given by the developer in the source code. However, the JavaScript language does not require names for many program artifacts, and, as we shall see, *nameless* or *anonymous* artifacts are more common than named ones. Anonymous artifacts prevent tools from communicating effectively with developers.

Among program artifacts, functions are central to understanding a JavaScript program. In addition to their role in the execution stack, they are first-class objects that are used for different purposes by developers; they might be used as an object constructor, a closure scope (module), or even passed as an argument in a function call. These functions can be defined and created without a name or identifier.

In this paper, we analyze 10 large, well-known JavaScript projects. We show that within these projects, less than 7% of the function bodies are named. We analyze the syntactic constructs surrounding these function bodies and rationalize how developers think about the bodies in

[1]As an evidence, JavaScript is used in all the Web’s 100 most popular sites [Richards2011].

relation to the structure of the code.

We propose an automated approach based on extracted data from the source code for naming JavaScript functions. The candidate function names can be used in debuggers for more descriptive object summaries and call-stack views, or in integration with proposed JavaScript typing systems for providing modern editing features in development environments.

7.1 The Anonymous Function Problem

A JavaScript function can be defined with the `function` definition or the `Function` constructor (i.e., `new Function( args, source)`) [ECMA1999]. The `function` definition can appear in a function declaration, function expression, or function statement. Of these forms, only the function declaration requires a function name and the `Function` constructor has no mechanism to name the function. In other words, JavaScript developers can define functions with or without function names.

If you are unfamiliar with JavaScript or other functional programming languages, you might imagine that developers would naturally select the form with names, simply as an organizational tool. However this is not the case. Functions in JavaScript are first-class objects. They can be assigned to any variable or object property, or passed as an argument to a function. Consequently JavaScript programmers can use these other constructs to organize their thinking about the program, without the use of function names.

So what do JavaScript developers do in practice? To get empirical evidence we analyzed the source code of ten well-known JavaScript projects[2]. For every project, the total number of functions and the number of functions with an identifier are shown in Table 7.1. The average ratio of named functions to all functions is less then 7 percent and, excepting one project, *Prototype*, the ratio does not exceed 13 percent. Among all functions only a very limited number of them (116 functions) are defined by the `Function` constructor. Our analysis does not include the functions defined dynamically by `eval` function or `new Function()`; these cases would only make the ratio even smaller. Therefore, we conclude that a large proportion of JavaScript functions are anonymous.

To understand the consequences of anonymous functions on development tools we will focus on one example, the impact on debuggers. Two main issues appear in debuggers due to the lack of function name. First, the object constructor name, which can facilitate understanding the object value, is not available in the object summary. Second, the call-stack view is usually full of *anonymous* functions and therefore much less informative. We discuss these issues in the next two subsections.

[2]We did not perform any preprocess to exclude third-party or repeated files in the provided source code bundles.

Project	Ver.	Description	Total	Named
Closure	r683	Google Web Library	9195	208(2%)
DoJo	1.5	JavaScript Toolkit	18676	2810(15%)
ExtJS	3.3.1	JavaScript Framework	37717	1184(3%)
Firebug	1.7	Web Development Tool	3424	406(11%)
jQuery	1.4.4	JavaScript Library	422	23(5%)
MochiKit	1.4.2	JavaScript Library	1866	37(1%)
MooTools	1.3	JavaScript Framework	625	7(1%)
Prototype	1.7	JavaScript Framework	645	203(31%)
Scriptaculous	1.9	JavaScript Library	1092	208(19%)
YUI	3.3	Yahoo UI Library	22346	922(4%)
All		All Projects	96008	6008(6.3%)

Table 7.1: The total number of functions and the number of named (and percent named) functions in ten large JavaScript projects. See appendix B for the project citations.

7.1.1 Missed Constructor Name in Object Summary

JavaScript does not support classes, but objects can be created by constructors (`new` followed by a function call). A constructor is a regular JavaScript function. Once the `new` keyword is evaluated, an empty object, with the constructor prototype as its prototype, is created, then the new object is bound to `this` and the constructor is called. The role of constructor is to initialize the empty object. Unlike class-based object-oriented languages, the structure of the object may change during the object's lifetime [Richards2010]. Nevertheless, the constructor can still be useful in classifying the object most of the time. Debuggers employ this fact and display the constructor name in the object summary to facilitate the developer's understanding.

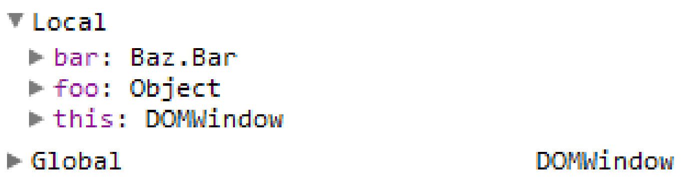
Figure 7.1 shows an excerpt of a JavaScript program we use to illustrate this issue. We set a breakpoint on line 15 and examine the runtime elements at this breakpoint by two JavaScript debuggers, Google Chrome and Firebug (Figure 7.2). Two objects assigned to variables `foo` and `bar` are constructed by two different constructors: `Foo` and `Bar`.

The Google Chrome debugger shows the general class of `Object` for `foo` and the `Baz.Bar` class for `bar` in their summaries. The first class is very general and the second one is misleading. The developer has to expand the object nodes to recognize their similarities and differences.

Firebug classifies both objects in the general `Object` class, but includes some of the object properties in the summary. These additional properties may give a hint to the developer about the object structures. `Foo` and `Bar` definitions at lines 20 and 25 explain the debuggers' behavior: the function statements has no explicit name (identifier), therefore debuggers considered them as anonymous functions or they infer a misleading name.

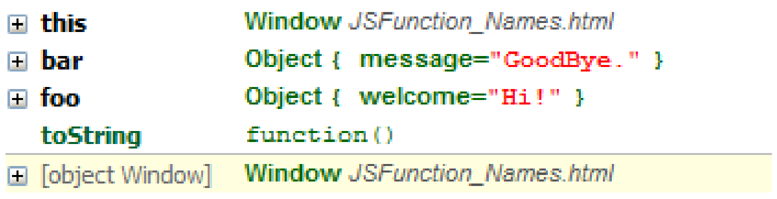
```
9   var main = function() {                  // main
10      var foo = new Foo(
11          function(){                      // main/foo<
12              this.welcome = "Hi!";
13          });
14      var bar = new Bar("GoodBye.");
15      alert(foo.welcome);
16      alert(bar.message);
17  };
18  var Foo = function(){                    // Foo<
19      var instances;
20      return function(initializer){        // Foo
21          instances++;
22          initializer.apply(this);
23      }
24  }();
25  var Baz = Bar = function(msg){           // Bar
26      this.message = msg;
27  }
```

Figure 7.1: An excerpt of a JavaScript code illustrating anonymous functions. The comments give the results from Sec. 7.4 and these are discussed in Sec. 7.4.5



```
▼ Local
  ▶ bar: Baz.Bar
  ▶ foo: Object
  ▶ this: DOMWindow
▶ Global                          DOMWindow
```

(a) Google Chrome Debugger



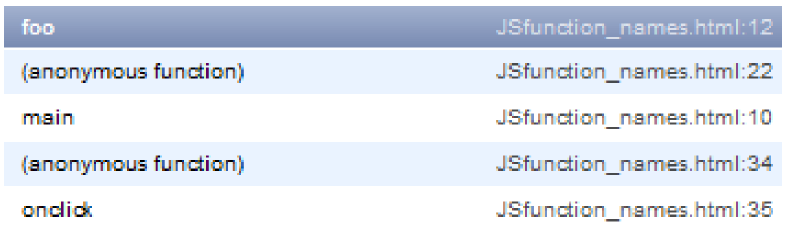
```
⊞ this              Window JSFunction_Names.html
⊞ bar               Object { message="GoodBye." }
⊞ foo               Object { welcome="Hi!" }
  toString          function()
⊞ [object Window]   Window JSFunction_Names.html
```

(b) Firebug

Figure 7.2: The screenshot of variables view of Google Chrome and Firebug JavaScript debuggers paused on a breakpoint at line 15 of the program shown in Figure 7.1. The content of these views is discussed in Sec. 7.1.1

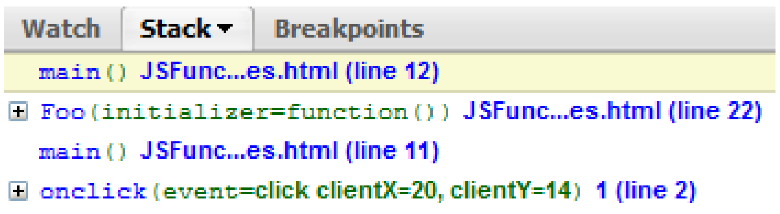
7.1.2 Anonymous Function Names in Call-stack View

The second problem is the call-stack. To illustrate this, we pause the program (Figure 7.1) at line 12 by a breakpoint. Figure 7.3 shows how the program call-stack is displayed in Google Chrome debugger and Firebug. The differences in the number of frames and line numbers between two call-stacks are due to dissimilar event handling implementations in the underlying platforms. Among the three top functions in the call stack, Google Chrome shows only the name of the third one down, function (`main`), correctly. For the second frame from the top (marked as line 22), it shows `anonymous`, and for the top frame (marked at line 12), it gives a wrong name, `foo`. Firebug performs better by guessing two function names correctly, but it still fails in one case. It shows `main` as the name of the function on the top frame (marked at line 12) but this is the name of another function (the enclosing function at line 9). In these cases, the information provided by the debugger is useless and the developer has to locate the function source to understand or recall the function behavior.



```
foo                          JSfunction_names.html:12
(anonymous function)         JSfunction_names.html:22
main                         JSfunction_names.html:10
(anonymous function)         JSfunction_names.html:34
onclick                      JSfunction_names.html:35
```

(a) Google Chrome Debugger



```
main() JSFunc...es.html (line 12)
⊞ Foo(initializer=function()) JSFunc...es.html (line 22)
main() JSFunc...es.html (line 11)
⊞ onclick(event=click clientX=20, clientY=14) 1 (line 2)
```

(b) Firebug

Figure 7.3: The screenshot of call-stack view of Google Chrome and Firebug JavaScript debuggers paused on a breakpoint at line 15 of the program shown in Figure 7.1. The contents of these views are discussed in Sec. 7.1.2

7.2 Automated Function Naming

The anonymous functions problem is discussed in several articles and forums on the Web [DisplayName, Zaytsev2010]. Different solutions have been proposed and discussed by practitioners. A basic solution is a mechanism for naming functions by developers without affecting the variables in scopes. For example, a new property (e.g., *displayName*) in the function object can be used for storing the function name, or the function name can be defined by an annotation. Although these solutions may help, they require extra work from developers, the displayName value can become out of sync with the meaning of the code over time, the annotation may be incorrectly recognized in programs that use the same property name for another purpose, and maintenance of the debugger becomes more difficult once the call-stack names can be overridden by user code.

We instead propose an automated approach for naming anonymous functions by analyzing the source code. Before getting into explaining the algorithm details we discuss the rationale behind some of decisions we made in this approach.

7.2.1 What Should Be Named?

A statement defining a function creates a new `Function` object. The definition may be evaluated multiple times, and depending on the times a function definition is evaluated, zero to many `Function` objects can be created from the same definition. Two function objects which are created from the same function body may have different object properties added at runtime. They may also have different enclosing scopes and therefore different behaviors. Thus our first question: do we try to name the `Function` objects or the source that defines them?

For the common cases, the different `Function` objects are bound to one or more properties of objects. The names of these properties inform the developer about the role of the function in the actions of the object. To determine the actions of the functions in turn, the developer must read the function source (or perhaps its documentation). Our function names serve to recall or summarize that source or documentation for the developer. Therefore we seek to name the source, the content between the curly braces known as the *FunctionBody* in the standard[ECMA1999].

After reflection the reader may be puzzled by the preceding claim. On the one hand we claim that the function object instance may be bound to properties in multiple objects and those property names are not helpful for naming. On the other hand we will shortly introduce an algorithm that uses a property name (in part) to name a *FunctionBody*. Ultimately we are relying on a subtle characteristic of JavaScript programming: the first binding of a *Function* to an object property differs from all other bindings because it is located in text near the *FunctionBody* and thus developers associate this first binding with the meaning of the *FunctionBody*.

7.2.2 What Makes a Good Function Name?

A function name is basically used to assist the developer to recall or understand the function behavior. For a developer who is already familiar with the function, it works more like an identifier. However, this identifier should be easily recognized by the developer. For example, a naive proposal for the function name is a combination of the function file name and its first line number. Although it may work as an identifier, it does not assist the developer to recall or understand the function behavior.

On the other hand, for a developer who does not know the function, a function name should explain the function behavior, or an abstraction of the function behavior, or why/where the function is used/defined (e.g., to create the object *foo*). A function name must not be so long that it can not be displayed or read by the developer. For example, the entire function body source code explains the function behavior well, however it is not an appropriate function name.

7.2.3 Context, Package and Function Names

JavaScript does not support a standard packaging mechanism to be used for modular programming. Scripts are loaded from different files and executed within the same or different global objects. Developers usually use objects at the top level to encapsulate objects, properties and functions from a framework or library. The same mechanism is reused for defining subpackages. As the project size and the number of functions increases, the short function name will not be enough for recognizing the function. The developer also wants the class or the module that contains the function. In addition to the package name, knowing the context (the enclosing function body) that contains the function can help in better understanding the function behavior.

7.3 Building Up Intuition by Example

We know that we face an ill-defined task: we are after all attempting to create short useful names for nameless functions. We have to create salient information from source code: we anticipate that removing characters will be our biggest challenge. To create an algorithm we decided to study the spectrum of examples from our 10 large collections of functions. We want to see what kinds of cases are important and what aspects of these cases help us identify functions. For this purpose we created 12 categories of function body expressions shown in Table 7.2 and we categorized all of the nameless functions from the 10 JavaScript projects into one of these 12 cases, giving the numerical results in Table 7.3.

Then we examine each of these cases to think about how we want the functions named. We will skip over the nesting of function scopes in the analysis to avoid taking on too much at one time; we return to this aspect at the end of this section.

7.3.1 Case 1: Object Property Initializer

This case usually appears when developers try to group a set of functions in a new object. A common case is grouping a set of functions in the `prototype` property of the constructor. This structure resembles the class structure in traditional object-oriented languages. When a new object is created by the constructor, the new object also inherits all functions defined in the constructor's prototype. This structure is also used when the owner object is a shared

	Description				Code
1	The function object	is assigned to a(n)	property of a new object in an object literal.		{ ..., foo: function(){...}, ...}
2			new array index in an array literal.		[..., function(){...}, ...]
3			object property through	direct access by a property identifier.	bar*.foo = function(){...}
4				hashmap access by a string.	bar*["foo"] = function(){...}
5				hashmap access by a variable name.	bar*[foo] = function(){...}
6				hashmap access by a JavaScript expression.	bar*[foo*] = function(){...}
7			array index.		foo*[0] = function(){...}
8			variable.		foo = function(){...}
9		is directly called.			function(){...}()
10		property is accessed.			function(){...}.foo
11		is returned from a function call.			{... return function(){...}}
12		is passed as an argument to a function.			foo*(..., function(){}, ...)

Table 7.2: Different cases of anonymous function object creation and usage in JavaScript. Identifiers with a star in the table can be expressions as well as simple identifiers; we explain how we reduce expressions to pseudo-identifier in 7.4.3.

	1	2	3	4	5	6	7	8	9	10	11	12
Closure	23(0.3%)	0	8466(94%)	3	4	0	0	67(0.7%)	16(0.2%)	0	43(0.5%)	365(4%)
DOJO	9601(61%)	7	1765(11%)	21	24	2	1	1151(7%)	476(3%)	2	175(1%)	2641(17%)
ExtJS	30221(83%)	0	1476(4%)	3	40	9	0	859(2%)	788(2%)	180	517(1%)	2439(7%)
Firebug	2296(76%)	0	539(18%)	1	2	0	0	17(0.4%)	7(0.2%)	2	6(0.2%)	148(5%)
jQuery	233(58%)	0	34(9%)	0	10	2	0	24(6%)	10(2%)	0	0	86(21%)
MochiKit	1080(59%)	10	385(21%)	0	4	0	0	110(6%)	18(1%)	0	41(2%)	181(10%)
MooTools	339(55%)	0	79(13%)	0	4	3	0	53(9%)	21(3%)	20	14(2%)	85(13%)
Prototype	265(60%)	0	28(6%)	0	0	1	0	25(6%)	44(10%)	1	8(2%)	70(16%)
Scriptaculous	564(64%)	0	75(8%)	0	2	2	0	27(3%)	45(5%)	21	9(1%)	139(15%)
YUI	14154(66%)	7	1721(8%)	0	90	0	0	1181(6%)	172(1%)	0	95(0.5%)	4004(19%)
All	58776(65%)	24	14568(16%)	28	180	19	1	3514(4%)	1597(2%)	226	908(1%)	10072(11%)

Table 7.3: The number of nameless functions in each category defined in table 7.2.

object with a set of utility functions.

The majority of nameless functions (more than 65%) in almost all studied projects (except Closure), are defined in object literals. This case seems particularly simple: the property name makes a good name for the function body. But what logic are we implicitly applying here? Our reasoning: the developer-invented property name has high information content, it is textually close to the function body, and the function object created by the function body initializes to the property. These observations guide us in more complex cases.

7.3.2 Case 2: Entry in an Array Literal

Contrary to the previous case the appearance of this case is very limited. It usually appears in initializations or when an array of functions are passed as an argument. Among all projects only three have instances of nameless functions defined in this way. For example array argument to `Event._attach` in this example from YUI [3]:

```
_attach: function (el, notifier, delegate) {
    if (Y.DOM.isWindow(el)) {
        return Event._attach([type, function (e) {
            notifier.fire(e);
        }, el]);
    }
}
```

The developer will probably think of the array entry as one item in a collection passed to `Event._attach`. In general we shall want to name these kinds of functions by the destiny of the containing array.

7.3.3 Case 3: Property Assignment With Property Identifier

This case is the second most common case in the studied projects. Here, we can see why the Closure project is different from other projects in the first case. About 94% of nameless functions in this project are defined in this way. It seems that the Closure developers follow an internal standard for function objects creation and usage.

Following the model from Case 1, we think a good name would combine the object name with the property identifier. The complication in this case comes from the object name: in general the object reference can be a computed expression[4]. Here is a simple example from the Closure project:

[3]The example is nested in more function definitions we do not show here.

[4]This comment applies to all of the cases in table 7.2 marked with an asterisk on the expression identifier

```
this.eventPool_.createObject = function() {
    return new goog.debug.Trace_.Event_();
};
```

In general, the expression can be long and complex: to create a useful name we need to focus on developer-invented identifiers in the expression and work to keep the total number of characters small. For example, `this.` add no information to the name since we cannot know the value of `this` while parsing.

7.3.4 Case 4: Property Assignment With Property Name String

In JavaScript, objects are like hashmaps and their properties can also be accessed by a string specified in the brackets after the object. Semantically this is the same as the previous case and we see few instances of this form of function object assignment. The string inside the brackets can be considered a property identifier for naming.

7.3.5 Case 5: Property Assignment With Property Name Variable

Object member names can be variable references that get converted to strings at runtime: this is syntactically similar to Case 4, but we cannot (usually) statically compute the string to use as a property identifier. The usage of this case is also limited. This form usually appears when the same function body is assigned to different properties in a loop. For example see the inner function in Fig. 7.4. The variable name in the cases we examined was a generic name like `o` or `item`. Unlike the previous two cases, we do not have a specific property name. Nevertheless identifying the function body using the assignment target with the variable name as the property name follows the reasoning used for the simple cases.

```
jQuery.each("ajaxStart ajaxStop ajaxComplete ajaxError".split(" "),
    function( i, o ) {
        jQuery.fn[o] = function( f ) {
            return this.bind(o, f);
        };
});
```

Figure 7.4: An example of a function (the inner definition) assigned to a hashmap using a variable name (row 5 in Table 7.2) and an example of functions passed as arguments to a function (row 11 in Table 7.2). The function is from the jQuery library but simplified to fit on the page.

7.3.6 Case 6: Property Assignment With Property Name Expressions

Object member names can be expressions that get converted to strings at runtime: this is more general than case 5. A common case of expression in this case is a conditional expression, e.g., `condition?"prop1":"prop2"`, where the property that we assign the function to depends upon runtime values. Another kind of example of computed names comes from the Prototype project (reformatted to fit in the page):

```
function define(D) {
    if (!element) element = getRootElement();
    property[D] = 'client' + D;
    viewport['get' + D] = function() {
        return element[property[D]]
    };
    return viewport['get' + D]();
}
```

The word `get` is concatenated with the `toString()` value of the argument `D` at runtime to create the property name. Unlike Case 5, the property name expression need not be a simple developer-invented identifier. In this example, `viewport[getD]` could be a good name, but in general we will need to process the expression to balance length with information.

Notice that from the programming language point of view, Cases 3 through 6 are all special cases of Case 6. After all we are just selecting an object property in all of these cases. But from a naming point of view these cases present different challenges and the more complex cases will make our necessary tradeoffs more costly.

7.3.7 Case 7: Assignment to an Element of an Array

We only observed one instance of this form in the studied projects. The numerical index should clearly be part of the name; the array name may be an expression that we have to analyze to create name.

7.3.8 Case 8: Assignment to a Variable

This case is widely used and we expect developers would expect the function body to get the name of the variable. As the function objects' bodies are immutable, it is very likely that a function object which is assigned to a variable, is used with the same variable name in the function scope and its internal scopes. There are cases in which a variable name is used temporarily, as the function is passed to another function or assigned to an object property. However, in most cases the variable name works well as the function name.

7.3.9 Case 9: Anonymous Functions Immediately Called

Calling function objects just after their creations is a common pattern in JavaScript. For example see the assignment to `Y.ClassNameManager` in Fig. 7.5 (the function is called at the bottom of the example).

```
Y.ClassNameManager = function () {
    var sPrefix = CONFIG[CLASS_NAME_PREFIX],
    sDelimiter = CONFIG[CLASS_NAME_DELIMITER];
    return {
        getClassName: Y.cached(function () {
            var args = Y.Array(arguments);
            if (args[args.length-1] !== true) {
                args.unshift(sPrefix);
            } else {
                args.pop();
            }
            return args.join(sDelimiter);
        })
    };
}();
```

Figure 7.5: An example the YUI project of function bodies from cases 9 (the outer function) and 12 (the argument to Y.cached) from Table 7.2

If the called-function is assigned to a variable or object property, then we have a version of one of the other cases in Table 7.2. The difference here is that we can tell from static analysis that the assignment will use the return value of the function, not the function object itself. But for naming purposes the key information will be the assignment target.

If the function call has no result, it means that the function performs one task (e.g., initialization of some values in the outer scope for later use). In this case we cannot use the assignment target idea from Case 1, but the source proximity and the developer-invented names concepts point to using interior identifiers in a name. To avoid confusing the developer by using the same name for the outer and interior functions, we will need some way to signify that the name we create in this way is for an immediately called function.

7.3.10 Case 10: Function Property is Accessed

In this unusual case a property of the function is accessed directly from the function body. This case usually happens when one of the predefined functions (i.e., `call`, `apply`, `bind`) or added functions to the `Function` prototype is called. Here is an example, from ExtJS, reformatted for display here:

```
setVisible : function(v, a, d, c, e){
    if(v){
        this.showAction();
    }
    if(a && v){
        var cb = function(){
            this.sync(true);
            if(c){
                c();
            }
        }.createDelegate(this);
        }
    ...
}
```

The inner function body is not used directly, but the result of `createDelegate` is assigned to `cb`. The most valuable information here is the variable name `cb`, followed by the `createDelegate` function name. This example also illustrates that automatic naming could have an effect on developers coding style: giving a longer name for `cb` would give better names in development tools but currently developers have limited expectations that tools will show such information.

7.3.11 Case 11: Returned From a Function Call

In this case the function body appears in a return statement of another function body. Although this category only includes 2% of nameless functions, proper naming of functions in this class is important. Many constructors are built using this form and therefore the names of these functions appear in object summaries. Clearly the name of these returned functions is almost the same as the name of the functions that define them. For example, in Fig. 7.6 a developer might pick names like `registerWinOnIE` and `registerWinNotOnIE` for the functions returned by the Dojo function `registerWin`.

7.3.12 Case 12: Function Passed as an Argument

Numbers in table 7.3 show that creating and passing functions as arguments is very common in JavaScript. For example, see Fig. 7.4. The calling function and the other arguments look helpful, to the extent that they have identifiers invented by the developer. As in this example, we see that the calling function, `jQuery.each()`, can be generic so it provides less valuable information, but the arguments in that example are highly specific to the function body. Fig. 7.5 shows different example, where the function called (`Y.Cached()`) seems much less important for naming the function than the property that we initialize with the result of calling the function. This important case will stress any naming algorithm: we somehow have to summarize the function call – which itself may be an expression – and the other arguments – any or all of which may be expressions.

```
registerWin: function(targetWindow, effectiveNode){
    ...
    if(doc){
        if(dojo.isIE){
            ...
            return function(){
                doc.detachEvent('onmousedown', mousedownListener);
                doc.detachEvent('onactivate', activateListener);
                doc.detachEvent('ondeactivate', deactivateListener);
                doc = null;
            };
        }else{
            ...
            return function(){
                doc.removeEventListener(
                     'mousedown', mousedownListener, true);
                doc.removeEventListener('focus', focusListener, true);
                doc.removeEventListener('blur', blurListener, true);
                doc = null;
        };
}}},
```

Figure 7.6: An example of a function body in a return statement, case 11 of Table 7.2 adapted from the Dojo project code

7.3.13 Results of Studying Examples

We reached two main conclusions from studying the way anonymous functions are used in the source of the 10 projects we examined. First, we want to try to find the name of the initializer or assignment target that will receive the function object created from a function body. JavaScript programmers are creating anonymous functions but they are loading them into object references and the expressions that result in those references have informative identifiers inside. Second, these expressions that we focus on may often be simple identifiers, but if they are not we will need to analyze the source code of these expressions to extract meaningful summaries. This summary has to balance information against length.

Overlaying our analysis above is hierarchy: any of the cases can be nested in function scopes. Obviously this hierarchy must be represented in our names. Algorithmically this is straight forward recursion. But from the name usability point of view, deep hierarchy means long names, exactly the problem we want to avoid. Fortunately developers are well trained in dealing with this kind of problem and we anticipate that hierarchical names can be shown to users in progressive depth depending on the particular needs in the user interface.

7.4 Static Function Object Consumption

Using our analysis we have created a preliminary automatic naming solution. The three parts of our solution match three observations for our study. First we apply *(Static) Function Object Consumption*, which tracks the function object created from a function body to where the object is 'consumed', for example by assignment to or initialization of an object property, variable reference, or function argument. The "static" qualifier just indicates that we will only use a parser. Second, we reduce complex expressions to pseudo-identifiers focusing on developer-invented names. Third we apply our approach hierarchically to deal with nested function bodies.

We will describe the details in the next sections. In constructing names we realized one additional aspect: as we move from simple object property names to more complex examples, the path of the object consumption is an added bit of information helpful in naming. Thus we add some symbols to guide the developer to the function body in complex cases: in this way the extra information does not take a lot of space and it can be ignored by developers who have not yet learned about its significance. We describe these symbols in Sec. 7.4.4.

7.4.1 Consumption Summary Algorithm

We parse the JavaScript and search the resulting syntax tree for function body nodes. For each function body, we apply the algorithm outlined in Algorithm 1. The basic idea (the `while` loop) is to walk the syntax tree from the body up through parent nodes until we hit a node that is not a JavaScript expression. For each node we create an entry in a list and record in it information about the relationship between the node and its parent. We'll use this relationship information in Sec. 7.4.4. Then for each node we record developer-invented identifiers related to the destiny of the function object created from the body as outlined in Table 7.4. For the first and third rows of the table we record the information recursively; for the second row, assignment node, we record the information after the loop terminates. The algorithm ends when we reach a node which is an assignment or a statement which does not return any value (i.e., the node is not an expression). The algorithm result is an *object consumption summary*, a list of collected data at every visited parent node of the function body.

The algorithm uses three subroutines: `getNextNode`, `nameExpression`, and `argSummary`. The `getNextNode` routine normally returns `n.parent`, but we also use this point in the algorithm to handle an important special case, function bodies inside of *immediate functions* typically used for modularity or scoping in JavaScript. We discuss this case in Sec. 7.4.2. The remaining two subroutines construct a pseudo-identifier from the expression as described in 7.4.3; they return their argument if it is simply an identifier.

Algorithm 1 Compute Object Consumption Summary for Function Body Nodes

```
Input: Function Body Node $n$ in Abstract Syntax Tree
Output: Object Consumption Summary
  $List\ summary = new\ List()$
  while $n.parent$ is an expression do
    $dataItem = new\ DataItem()$
    if $n$ value is same as $n.parent$ value then
      $dataItem.isSameAs = true$
    else if $n$ value is a property of $n.parent$ value then
      $dataItem.isPartOf = true$
    else
      $dataItem.isContributesTo = true$
    end if
    if $n.parent$ is a function call and $n$ is an argument then
      $dataItem.isFunctionCall = true$
      $dataItem.id = nameExpression(n.parent)$
      $dataItem.hint = argSummary(n.parent)$
    else if $n.parent$ is an object literal and $n$ is its expression then
      $dataItem.isObjLiteral = true$
      $dataItem.id = n.parent$ property name
    end if
    $summary.add(dataItem)$
    $n = getNextNode(n)$
  end while
  if $n.parent$ is an assignment then
    $dataItem = new\ DataItem()$
    $dataItem.isAssignment = true$
    $dataItem.id = nameExpression(n.parent)$
    $summary.add(dataItem)$
  end if
  return $summary$
```

	Description		Code
1	The parent node	is an object literal.	{ ..., foo: expr }
2		is a function call.	foo*(..., expr, ...)
3		is an assignment.	foo* = expr

Table 7.4: Nodes produce identifiers in the function object consumption summary. Identifiers with a star in the table can be expressions as well as simple identifiers; we explain how we reduce expressions to pseudo-identifier in 7.4.3.

7.4.2 Consumption by Immediate Functions

JavaScript developers use function scope to dynamically create functions with shared but private state. In this pattern, an enclosing function contains a number of function and object definitions and it is called immediately after it is defined. We call these functions *immediate functions.* For an example, see line 10 of the Figure 7.1 which is enclosed in the function on line 9. If the developer returns a function from this outer, immediate function, we want to follow the returned object to where it is consumed.

To keep our core algorithm simple we handle these returns from immediate functions as a special case. At the end of each loop in Algorithm 1 at the point marked `getNextNode()` we check to see if the parent is a `return` node. If not we return the parent node and the loop continues. If we have a `return` node, we look to the parent of the `return` node to see if it is a `function` node with a parent call node (either directly with `()` or via `apply()` or `call()`). If so, we know we are returning a function object from an immediate function. We return the parent of the immediate function, effectively skipping the intermediate nodes so that the name will reflect the consumption of the return value into the destination of the immediate function.

7.4.3 Expression Reductions

In simple cases the syntax tree node will have an identifier we can use as part of our name. In more complex cases an expression will be written in place of an identifier. We reduce these expressions to pseudo-identifiers (i.e. not necessarily a valid JavaScript identifier) that resembles the expression. This work is done during the Object Consumption Summary algorithm in the functions described here:

Identifiers for Function Arguments Search for all literal string nodes in other arguments and concatenated them by “-”, dropping characters beyond 10. (This work is on in Algorithm 1 in `argSummary`).

Identifiers for Assignments and Function calls This function gets called for nodes resulting from the last 2 rows of Table 7.4. The expressions here will evaluate to a writable or readable address, so we want to extract the most specific developer-invented identifiers from the expression. Thus we apply the rules from Table 7.5, which starts from the right hand side of the expression; we skip any JavaScript keywords like `this` or `prototype`. We also skip any pattern which does not match (e.g., a function call). (This work is on in Algorithm 1 in `nameExpressions`.)

Obviously these rules are heuristic and can be improved through experience and interaction with developers. We are attempting to balance information content with length. Large complex expressions will give pseudo-identifiers which are complex, but with some identifiable parts

Description	Pattern	Name
Primitive	value	value.toString()
Variable	id	id
GetProp	e.id	Name(e).id
GetElem	e1[e2]	Name(e1)+[+Name(e2)+]
Operation	e1 op e2	Name(e1)+op+Name(e2)
Condition	cond?e1:e2	Name(e1)+:+Name(e2)

Table 7.5: JavaScript Expression Reduction to a Name. Expressions which match an entry in the pattern column are converted as shown in the Name column. Here `e` indicates an expression, `id` indicates an identifier, + means string concatenation and `Name()` means we apply the pattern matching recursively.

adequate for developer recognition and search.

7.4.4 Conversion to a Name

At this stage of the algorithm we have a list of identifiers with attributes which we want to concatenate to create a name.

First we drop function-call identifiers if we have anything else to use for a name. The function-call identifiers typically tell us about a transformation of the function body before it is assigned to an object property or variable. The transformation may be used many places in the code, while the assignment target is typically an identifier defined by the developer for a specific section of source code. See, for example, function on line 14 and the function call on line 13 in the Figure 7.1. Specifically we drop identifiers from row 2 of table 7.4 in any case where we have identifiers from rows 1 or 3. The outer function in Fig. 7.4 illustrates the opposite case, where we do not drop function-call identifier.

Second we concatenate the identifiers with a symbol between each parent and child showing the relationship. As shown in Algorithm 1, if the parent expression is an array or object literal then the identifier will be marked as *isPartOf* and we insert a dot character. If the identifier was marked as *isContributesTo* we insert a left angle bracket. An identifier marked *isSameAs* is skipped over because we already have identifier information for it. An identifier marked *isAssignment* suppresses any other marks to signify the importance of the assigned-to identifier. Because some entries on the object consumption list are empty strings we may have duplicate symbols. Any duplicates are replaced by a single symbol. The result contains identifiers (i.e., variable and property names) and strings available in the source code plus some explanatory tokens. It compactly explains the function object creation, consumption and assignment.

The particular symbols we use may be refined with more experience in how developers respond to them. Our intuition is that these extra symbols need to be visually compact because their purpose is to adjust the developers expectation for the identifier. For example, the function on line 11 of Fig. 7.1 is named `main/foo<` but the developer may only key on the word `foo` to recall the function body.

The name built by the above process does not contain any information about enclosing scopes. We call it *local name*. We get the function *full name* (that is a local name qualified by its position in the scope hierarchy) by adding the enclosing function full-name with a slash before the local-name, recursing through enclosing scopes. This full-name is the function body name.

7.4.5 Examples

To further explain our approach we now apply it to the JavaScript code presented in Figure 7.1. We can recognize five function bodies in the code, none of them has a name.

The first function on line 9 is assigned to the variable `main`. The parent node for the function body in the syntax tree will be an assignment, so in Algorithm 1 we skip the `while` loop and compute the `nameExpression` as `main`. Ultimately this becomes the name of the function.

The second function on line `11` is nested in the first one. It is passed as an argument to a constructor call on line 10 and the resulting object is assigned to `foo`. In Algorithm 1 we create two entries in the consumption summary, one for the function call with identifier `Foo` and one for the assignment with identifier `foo`. The first one gets marked with `isContributesTo`. Following the logic in Sec. 7.4.4, we drop the function call identifier but mark the name with < for contributes-to. The resulting local name is `foo<` and the full name includes the enclosing function name: `main/foo<`.

The third function on line 18 is called immediately after definition, on line 24. This means the function body in the syntax tree will have a parent from the immediate call and then the assignment parent node; the call does not give us an identifier but it does give a consumption summary entry with `isContributesTo`. The name becomes `Foo<`.

The fourth function on line 20 is returned by `Foo<`. After the first pass through the `while` loop in Algorithm 1 we enter `getNextNode` and trigger the code described in Sec. 7.4.2. This will cause us to walk up the syntax tree to find the assignment target for the outer function. We end up with name `Foo`. Because do not process the intermediate nodes in the `while` loop we do not mark the name with contributes-to and we do not record that the function is nested. Of course the function body is nested, but it is bound to an un-nested variable. To keep the name compact we choose not to encode this complex information. Rather we stick to the simple picture that the function 'is' `Foo`. If the developer wants to know more they can look up `Foo` to see the construction and nesting.

The fifth function on line 25 is assigned to two variables. This example illustrates that we stop processing in Algorithm 1 as soon as we hit the first assignment, giving the name `Bar`. This aligns with our observation that the visually closest identifier is the best choice.

Finally, Table 7.6 gives the names of functions from examples discussed in 7.3.

Example Code	Static Function-Object Consumption Full Name
Sec. 7.3.2 outer	YUI.add(event-focus)/_attach
Sec. 7.3.2 inner	YUI.add(event-focus)/_attach/Event._attach()
Sec. 7.3.3	eventPool_.createObject
Fig. 7.4 outer	jQuery.each(ajaxStart)
Fig. 7.4 inner	jQuery.each(ajaxStart)/jQuery.fn[o]
Sec. 7.3.6	define/viewport[get+D]
Fig. 7.5 outer	YUI.add(classnamem)/Y.ClassNameManager<
Fig. 7.5 inner	YUI.add(classnamem)/Y.ClassNameManager.getClassName<
Sec. 7.3.10 inner	setVisible/cb<
Fig. 7.6 inner	registerWin/

Table 7.6: Results from applying the approach in Sec. 7.4 to the examples in the paper at the point given in the first column. Full names are listed, even in cases where our example code omitted the enclosing function scope.

7.5 Evaluation

In Table 7.7 we compared our results to Firebug's naming output for each of the 10 projects used previously and listed in Appendix 1. Firebug does not use a parser for naming functions. It instead employs a number of regular expressions and apply them on a few lines around the function definition to obtain a name (a function called `guessFunctionName()`). Although this approach is not reliable and may provide wrong names for some functions, it infers acceptable names in many simple cases. About 10 percent (the second column) of anonymous functions still remained nameless in Firebug (in addition, some named functions may be quite incorrect because of the simple algorithm). The third column shows that more than 98 percent of anonymous functions have a FOC-defined local name. By adding the enclosing function names, this number increases to more than 99 percent for FOC-Full(the fourth column). Based on our observations, most functions remaining anonymous in FOC-Full are top level *immediate* functions which are usually used for creating a local scope for a set of variables.

The "Duplicates By" columns show the number of functions which get a name which is also assigned to another function(s) for Firebug, FOC-Local and FOC-Full. The number shows that Firebug assigns duplicate names to about 47 percent of anonymous functions. The first reason behind this huge number is that Firebug relies on regular expressions instead of abstract syntax tree for locating names. The second reason is that Firebug does not analyze the function object flow but tries to construct the name from the identifiers close to function definition. The second column shows that FOC-Local gives much less duplicates (13%) comparing to Firebug. This means that local names are sufficient for recognizing functions within a file, which is helpful because they have shorter length than full names. The third column says that less than 9 percent of anonymous functions get duplicated names by full names. Based on our

Project	Anonymous Functions By				Duplicates by			Local/Full Length		
	Developer	Firebug	FOC-Local	FOC-Full	Firebug	FOC-Local	FOC-Full	Average	Longest	>50/>80
Closure	8987	312(3%)	51	10	662(7%)	211(2%)	103(1%)	33/35	76/101	471/30
DoJo	15866	3362(21%)	528	390	3325(21%)	2993(19%)	2287(14%)	24/36	81/162	680/765
ExtJS	36533	1543(4%)	631	231	23777(65%)	3737(10%)	2342(7%)	22/25	65/91	177/2
Firebug	3018	167(5%)	14	10	481(16%)	85(3%)	32(1%)	27/36	70/140	76/18
jQuery	399	95(24%)	9	8	162(40%)	97(24%)	56(14%)	14/18	50/65	0/0
MochiKit	1829	202(11%)	59	16	491(27%)	314(17%)	191(10%)	17/22	49/80	0/0
MooTools	618	121(19%)	30	18	272(44%)	158(26%)	138(22%)	12/14	36/73	0/0
Prototype	442	125(28%)	24	19	154(35%)	70(16%)	70(16%)	21/25	55/88	5/1
Scriptaculous	884	195(22%)	25	19	306(36%)	131(19%)	86(9%)	21/27	55/132	7/5
YUI	21424	3317(15%)	212	15	10073(47%)	3558(17%)	2205(10%)	16/37	114/145	223/292
All	90000	9439(10%)	1583(2%)	736(1%)	39703(44%)	11354(13%)	7510(8%)	N/A	N/A	1639/1113 (3%)

Table 7.7: Results of Function Object Consumption. The rows are the projects listed in Appendix 1. The first column contains the number of anonymous functions in each project. The second column shows the number of functions Firebug could not name. The third column contains the number of functions nameless after applying the static Function Object Consumption (FOC) algorithm; and the fourth column is the number of functions the FOC leaves without a name even from enclosing scopes. The next three columns give the number of times a Firebug, FOC-Local, and FOC-Full function name appears twice in a file, respectively. The last three columns contain the length information. Every cell has two entries divided by a slash, the first is the local name and the second is the full name. The eighth and ninth columns give the average and longest name character counts respectively. The last columns shows the number of functions with local names greater then 50 characters in length separated by a slash from the number of functions wit full names with length greater than 80 characters.

observations, many of duplicates in projects such as DOJO, ExtJS and YUI come from test files. In a test file the same use case is repeated with different data and therefore the same function names appear several times. In projects like JQuery, Prototype, MooTools the conditional statements are the main source of duplicates. These libraries usually check against different environment properties (e.g., browser) to load the appropriate function.

The last two columns show the average and longest function name lengths (in characters) for both short and full names. The average length is at most 37 characters. The last column shows that only 3 percent of functions get local names longer than 50 characters or full names longer than 80 characters. Again, based on our observations a main source of long names are deep nested functions which mostly appear in test files.

7.6 Discussion

Overall the Function-Object Consumption approach dramatically reduces the number of functions that a development tool cannot name for a developer. By looking at the examples and comparing the algorithms, the names selected by FOC will be the same as the one's selected by Firebug's `guessFunctionName()` when that function gives reasonable results. In other cases FOC will give a more useful name including many cases where the current Firebug approach fails.

There is room for improvements. The heuristic elements of the naming solution need to be tuned based on more experience. Given that our goals trade precision for rapid recognition, our names are not unique. We also leave some cases anonymous which need further investigation.

A complete naming solution will need to consider some additional issues. The user interface that shows names typically has limited space and even our efforts to create short names may be adequate. Often the hierarchy can help: we can show the trailing entry in a slash delimited list and the trailing entry in a dot delimited list with mouse-over expansion to bring up an overlay line with more of the full name displayed. Some JavaScript libraries have a regular pattern or specific re-naming registration system (see for example the ExtJS ClassManager[ExtJSClassManager]). A naming solution should recognize these patterns or systems. Similarly JavaScript libraries might support the `.displayName` to provide custom names where the developer overhead to introduce and maintain the extra field is justified by the wide spread use of the library.

7.7 Related Work

To best of our knowledge, this paper is the first study in naming anonymous functions in JavaScript. However, there have been a few studies on the JavaScript programs behavior. In a recent work, Richards et al. conducted a study on dynamic behavior of JavaScript programs

[Richards2010]. Although their study contains some aspects of JavaScript function objects creation and usage, such as the number of different call sites and arities in function calls, it does not provide any data about anonymous functions.

We previously mentioned four JavaScript naming approaches: Zaytsev advocates expanded use of names in function expressions[Zaytsev2010], support for `displayName` in debuggers has been requested[DisplayName], the Firebug Regular expressions, and the Google Chrome debugger's function name 'inference'[GoogleChromeCode]. The last one appears to be a parser based approach with goals similar to the ad-hoc Firebug regular expressions: look for identifiers preceding anonymous function bodies.

Høst and Østvold [Host2009] attempted to find and fix inappropriate function names in Java programs. Their approach employs rules extracted from a large corpus of Java projects to recognize buggy names. To fix a name, a list of candidate names are constructed and ranked, the name with the highest rank is proposed for the replacement. This approach assumes that the function is already named and constructs the names based on the function internal structure. In contrast, our approach construct the function name by analyzing the function context.

Caprile and Tonella [Caprile1999], analyze the structure of function identifiers in C programs. The identifiers are decomposed into fragments that are then classified into seven lexical categories. The structure of the function identifiers are further described by a hand-crafted grammar.

The result of our work can be improved and impact other aspects of JavaScript programming if a stronger typing system is employed. A few static typing systems have been proposed for JavaScript [Anderson2005, Anderson2005.2, Heidegger2009, Thiemann2005]. These approaches discover the type of values and object structures for a variable by statically analyzing the source code and possible program control flows lead to the variable assignment. The discovered facts about variable types are not only useful for catching errors but to assist developers in code comprehension. Nevertheless, none of the mentioned approaches provided effective means for sharing this information with the developer. Our approach can help in this regard by assigning names to nameless elements.

7.8 Summary

Our contributions in this chapter include an empirical study of the extent of anonymous functions in JavaScript libraries, categorization of those functions to understand the potential for automatic naming, a practical algorithm based on the empirical study, and its evaluation. We believe our result can be applied directly in existing JavaScript development tools to give immediate benefit to developers and provides a basis for future improvements.

Debugging is anticipated with distaste, performed with reluctance, and bragged about forever.

— Anonymous

8 Conclusion

In this dissertation, we introduced *querypoint debugging* as a new approach to buggy execution inspection. We showed the feasibility of this approach by implementing a querypoint debugger. At the core of this approach are querypoints. We explained three basic category of querypoints: controlled forward navigation, causlity backward navigation, and search. Among these three, we mainly focused on causality backward navigation and identified three basic querypoints in this category: *lastChange*, *lastCondition*, and *valueOrigin*. We implemented and integrated *lastChange* to a traditional debugger. Now, what are the next steps?

8.1 Future Work

We believe that querypoints are practical and can be added to current debuggers. Perhaps, the best approach for identifying the next steps is receiving feedback from developers after supporting a basic querypoint like *lastChange*. However, we speculate the potential directions here.

8.1.1 *lastChange* on Complex Structures

We explained *lastChange* on alias and expression values. But, what about objects with complex structures such as hash tables. In JavaScript, a hash table keys are like object properties, so *lastChange* can be defined and processed on the keys of a hash table similar to regular properties. In Java, however, hash tables are different. In order to process *lastChange* on a hash table key, we need to instrument `put` methods. Providing mechanisms for defining a new, customized *lastChange* on a complex structure is important for both debugger developers and users.

8.1.2 Controlled Forward Navigation

The main feature of querypoints in this category is re-use. Re-use of step, step summary, and assertions for debugging new bugs. In chapter 4, we briefly explained how these items should be defined. Here, we discuss more details.

Identifying Data-Flows

We mentioned that data-flows are at the center of control forward navigation. However, identifying data-flows is hard. First, they are not explicitly defined in most programming languages. Second, they are not well-defined and dependent on the way a developer understands the execution.

One of the challenges to address is the cases where the data-flow forks or goes to another thread (by asynchronous or synchronous calls) or goes out of debuggee boundary (Again. by asynchronous or synchronous calls). Most of these calls are not defined in the language but in libraries and, this makes identifying them even more difficult.

Defining Steps

We can consider two groups of steps. First, the steps that can be identified from the source code or execution structure. For example, a loop iteration. Second, program-dependent steps which are specific to the buggy program. For example, a step where an email is received in an email client application.

The first group of steps can be identified and proposed by the debugger. However, for the second group of steps, we need visual interactive mechanisms that require little effort for defining new steps and their summaries.

A *lastChange* User Study

A.1 Introduction Page

lastChange **User Study**

Thank you for participating in this user study. The goal of this user study is to assess the usability of a new functionality, lastChange, in Firebug.

Through this user study which takes about 45 minutes you must debug two sets of buggy programs and report the defects.

Please go through the following setup steps:

1. Make sure you have installed Firefox 3.6.X on your computer.
2. Create a new Firefox profile for this user study and open Firefox with this profile.
3. Install Firebug 1.7X.0a5 .
4. Install Querypoint extension.
5. Close and re-open Firefox.
6. Open Firebug (F12 or Firebug Icon), then open Firebug Tracing panel, "Shift-R"
7. For the breakpoints part of the study please use the "replay" button on the Firebug Script panel
8. At the end of the study, please use "Tools > Save to File" on the Tracing window and email us the .ftl file.
9. Click on your group.

   Group 1

   Group 2

A.2 Group 1's Instructions

Group 1 - ***lastChange*** **User Study**

Please, go through the following steps:

1. Study lastChange.

   Introduction to lastChange

   Screenshots of the Querypoint user interface

2. Practice lastChange.

   Basic

   (optional) Salary Calculation

   (optional) Graph Traversal

3. Debug a program using only breakpoints (Time Trial). Use the replay button on the Script panel if you want to start execution over.

   Shapes

4. Debug a program using lastChange (Time Trial).

   Moving Circle

Thanks! Please remember to send us the tracing window Tools > Save to File results.

A.3 Shapes Page

Shapes

This page calculates Perimeter and Area for a list of shapes. Go through these steps:

1. Click on the button.

2. When you see the error in Firebug console, set a breakpoint on the error (red dot), then re-run to halt on the error message.

3. Locate the defect causes the error.

A.4 Shapes Source Code

```
<html><head><title>Shapes</title><script type="text/javascript"><!--
            var compute = function(){
                console.log("start .............");
                var shapes = getShapes();
                for (var i=0 ; i<shapes.length ; i++){
                    perimeter(shapes[i]);
                    area(shapes[i]);}
                shapes.sort(function(a,b){return a.type>b.type;});
                for (var i=0 ; i<shapes.length ; i++){
                    checkShape(shapes[i]);}
                console.log(".............  end.");};
            var perimeter  = function(shape){
                if (shape.type === "circle"){
                    shape.perimeter = 2*Math.PI*shape.radius;}
                if (shape.type === "rectangle"){
                    shape.perimeter = 2*(shape.length+shape.width);}
                if (shape.type === "square"){
                    shape.perimeter = 4*shape.length;}
                if (shape.type === "triangle"){
                    shape.perimeter = shape.base+shape.side1+shape.side2;}
                if (shape.type === "Parallelogram"){
                    shape.perimeter = 2*(shape.base+shape.side);}};
            var area = function(shape){
                if (shape.type === "circle"){
                    shape.area = Math.PI*shape.radius*shape.radius;}
                if (shape.type === "rectangle"){
                    shape.area = shape.length*shape.width;}
                if (shape.type === "square"){
                    shape.area = shape.length*shape.lnegth;}
                if (shape.type === "triangle"){
                    shape.area = shape.base*shape.height;}
                if (shape.type === "Parallelogram"){
                    shape.area = shape.base*shape.height;}};
            var checkShape = function(shape){
                if (!shape.perimeter || !shape.area)
                    throw new Error("Perimeter or Area has not been been calculated");};
            var getShapes = function(){
                var shapes = [];
                var shape = {type:"circle", color:"blue", radius:5};
                shapes.push(shape);
                var shape = {type:"square", color:"red", length:3};
                shapes.push(shape);
                var shape = {type:"rectangle", color:"red", length:3, width:4 };
                shapes.push(shape);
                var shape = {type:"triangle", color:"yellow", base:3, height:4, side1:4, side2:5};
                shapes.push(shape);
                var shape = {type:"Parallelogram", color:"blue", base:3, height:4, side:5};
                shapes.push(shape);
                var shape = {type:"rectangle", color:"red", length:3, width:4};
                shapes.push(shape);
                return shapes;};
        --></script>
    </head>
    <body id="myBody"><button id="reproducer" onclick="compute()">
      Calculate Perimeter/Area !
    </button></body>
</html>
```

A.5 Moving Circle Page

Moving Circle

This page shows a moving circle. Go through these steps:

1. Click on the button.
2. When you see the error in Firebug console, set a breakpoint on the error (red dot), then re-run to halt on the error message.
3. Locate the defect causes the error.

A.6 Moving Circle Code

```
<html><head><title>Moving Circle</title><script type="text/javascript"><!--
            var circle = {color:"#FF1C0A"}; var vector = {};
            var defaultValues = {x : 100, y:100, r:10};
            var compute = function(){
                console.log("start .............");
                window.setInterval("drawCircle(getRandomVector())", 1000);};
            var drawCircle = function(vector){
                //get a reference to the canvas
                var ctx = document.getElementById("canvas").getContext("2d");
                if (circle.x!==vector.x || circle.y!==vector.y || circle.r!==vector.r){
                    //remove the old one
                    ctx.clearRect(0,0,canvas.width,canvas.height);
                    circle.x = vector.x;
                    circle.y = vector.y;
                    circle.r = vector.r;
                }
                if (circle.r === 0)
                    throw new Error("Error : Radius is zero, use the default value!");
                //draw a circle
                ctx.fillStyle = "#FF1C0A";
                ctx.beginPath();
                ctx.arc(circle.x, circle.y, circle.r, 0, Math.PI*2, true);
                ctx.closePath();
                ctx.fill();};
            var getRandomVector = function(circle){
                vector.x = Math.floor(Math.random()*800);
                vector.y = Math.floor(Math.random()*200);
                vector.r = Math.floor(Math.random()*5)*6;
                if (vector.r !== 0 && vector.r < 30);
                    return vector;
                vector.r = defaultValues.r;
                return vector;}
        --></script>
    </head>
    <body id="myBody">
    <button id="reproducer" onclick="compute()">Start !</button>
    <canvas id="canvas" width="800" height="200"></canvas>
    </body>
</html>
```

B JavaScript Projects Analyzed for Function Names

Closure Closure Library, r683, Google Code,
http://code.google.com/p/closure-library

Dojo Dojo JavaScript Toolkit, version 1.5,
http://dojotoolkit.org

ExtJS JavaScript Framework, version 3.3.1
http://www.sencha.com/products/extjs

Firebug Web Page Debugger, version 1.7
http://code.google.com/p/fbug

jQuery JavaScript Library, version 1.4.4
http://jquery.com

MochiKit version 1.4.2
http://mochi.github.com/mochikit

MooTools JavaScript Framework, version 1.3
http://mootools.net

Prototype JavaScript Framework, version 1.7
http://www.prototypejs.org

Scriptaculous JavaScript Library, version 1.9
http://script.aculo.us

YUI Library, Yahoo, Inc., version 3.3
http://developer.yahoo.com/yui

Bibliography

[Agrawal1990] H. Agrawal, and J. R. Horgan. Dynamic programming slicing. In *Conference on Programming Language Design and Implementation(PLDI)*, 1990.

[Agrawal1990] H. Agrawal, R. A. Demillo, and E. H. Spafford. An Execution-Backtracking Approach to Debugging, IEEE Software, pp. 21-26, May/June, 1991

[Agrawal1993] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. Softw: Pract. Exper., 23: 589–616 (1993).

[Allan2005] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. De Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications(OOPSLA)*, 2005.

[Anderson2005] C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. In *Proceedings of the 19th European conference on Object-Oriented Programming (ECOOP)*, July, 2005.

[Anderson2005.2] C. Anderson and P. Giannini. Type checking for javascript. *Electr. Notes Theor. Comput. Sci.*, 138(2), 2005.

[Araki1991] K. Araki, Z. Furukawa, and J. Cheng. A General Framework for Debugging. *IEEE Software*, v.8 n.3, p.14-20, May 1991.

[Balzer1969] R. M. Balzer. Exdams: extendible debugging and monitoring system. in *AFIPS proceedings, Spring Joint Computer Conference*, 1969.

[Barton2010] J. J. Barton, and J. Odvarko. Dynamic and graphical web page breakpoints. In *Proceedings of the 19th international conference on World wide web (WWW)*, 2010

[Bhansali2006] S. Bhansali, W. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, J. Chau. Framework for instruction-level tracing and analysis of program executions. In *International Conference on Virtual Execution Environments(VEE)*, June, 2006.

[Bodden2011] E. Bodden. Stateful breakpoints: a practical approach to defining parameterized runtime monitors. In *Proceedings of the the 8th joint meeting of the European software* *engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE)*, , 2011.

[Bond2007] M.D. Bond, N. Nethercote, S.W. Kent, S.Z. Guyer, and K.S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *22nd annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications(OOPSLA)*, October, 2007.

[Boothe2000] B. Boothe. Efficient algorithms for bidirectional debugging. In *Conference on Programming Language Design and Implementation(PLDI)*, June, 2000.

[Caprile1999] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the 6th working conference on Reverse Engineering(WCRE)*, October, 1999.

[Chern2007] R. Chern, and K. D. Volder. Debugging with control-flow breakpoints. In *Proceedings of the 6th international conference on Aspect-oriented software development (AOSD)*, 2007.

[CommonJS] Common JS. http://www.commonjs.org

[Czyz2007] J.K. Czyz, and B. Jayaraman. Declarative and visual debugging in Eclipse. In *Proceedings of the 2007 OOPSLA workshop on Eclipse Technology eXchange(ETX)*, October, 2007.

[DisplayName] Add prettyName/displayName support to Profiler output and Stacks. *Firebug Bug Repository*, http://code.google.com/p/fbug /issues/detail?id=1811

[ECMA1999] ECMA International. *ECMA-262: ECMAScript Language Specification*, ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, December 1999.

[Eisenstadt1997] M. Eisenstadt. My hairiest bug war stories. *Commun. ACM*, 40(4):30–37, 1997.

[ExtJSClassManager] Jacky Nguyen. *Ext.ClassManager*. http://docs.sencha.com/ ext-js/4-0/#/api/Ext.ClassManager

[FirefoxAddons] Firefox Add-ons. https://addons.mozilla.org/en-US/developers /docs/getting-started.

[GoogleChromeCode] V8 Google FuncNameInferrer. http://v8.googlecode.com/svn /trunk/src/func-name-inferrer.h

[GoogleChromeExtensions] Google Chrome Extensions. http://code.google.com/chrome /extensions.

[Gu2010] Z. Gu, E.T. Barr, D.J. Hamilton, and Z. Su. Has the bug really been fixed? In *Proceedings of the 32th international conference on Software engineering (ICSE)*, 2010.

[Heidegger2009] P. Heidegger and P. Thiemann. Recency types for dynamically-typed, object-based languages. In *Proceedings of Foundations of Object Oriented Languages (FOOL)*, 2009.

[Hofer2006] C. Hofer, M. Denker, and M. Ducasse. Design and implementation of a backward-in-time debugger. In *Proceedings of NODE*, 2006.

[Host2009] E. W. Høst and B. M. Østvold. Debugging Method Names. In *Proceedings of the 23rd European conference on Object-Oriented Programming (ECOOP)*, July, 2009.

[Ingalls2008] D. Ingalls, K. Palacz, S. Uhler and A. Taivalsaari. The lively kernel a self-supporting system on a web page. In *Self-Sustaining Systems*, 2008.

[JS-MSOffice] JScript development in Microsoft Office 11. http://msdn. microsoft.com/en-us/library/aa202668(office.11).aspx

[JS-OpenOffice] JavaScript development in OpenOffice. http://framework. openoffice.org/scripting/release-0.2/javascript-devguide.html

[Ko2008] A.J. Ko, and B.A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering(ICSE)*, May, 2008.

[LaToza2006] T.D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering(ICSE)*, May, 2006.

[LaToza2007] T.D. LaToza, D. Garlan, J.D. Herbsleb, and B.A. Myers. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE)*, September, 2007.

[LaToza2010] T.D. LaToza, and B. Myers. Hard-to-Answer Questions about Code. In *Proceedings of the 2nd Workshop on the Evaluation and Usability of Programming Languages and Tools*, October, 2010.

[Lee2009] B. Lee, M. Hirzel, R. Grimm, K.S. McKinley. Debug all your code: portable Mixed-environment debugging. In *Proceedings of the 24th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications(OOPSLA)*, October, 2009.

[Lewis2003] B. Lewis, and M. Ducasse. Using events to debug Java programs backwards in time. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications(OOPSLA)*, 2003.

[Lienhard2008] A. Lienhard, T. Gîrba, and O. Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European conference on Object-Oriented Programming (ECOOP)*, July, 2008.

[Lienhard2009] A. Lienhard, J. Fierz, and O. Nierstrasz. Flow-Centric, Back-in-Time Debugging. In *Proceedings of the 47th International Conference on Objects, Models, Components, Patterns (TOOLS)*, July, 2009.

[Lencevicius2003] R. Lencevicius, U. Hölzle, and A.K. Singh. Dynamic Query-Based Debugging of Object-Oriented Programs. *Autom. Softw. Eng.* 2003.

[Martin2005] M. Martin. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications(OOPSLA)*, 2005.

[Maruyama2003] K. Maruyama, and T. Kazutaka. Debugging with Reverse Watchpoint. In *Proceedings of the Third International Conference on Quality Software*, 2003.

[Matthijssen2010] N. Matthijssen, A. Azidman, M. Storey, I. Bull, A.V. Deursen. Connecting Traces: Understanding Client-Server Interactions in Ajax Applications. In *Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC)*, July, 2010.

[Mayrhauser1997] A.V. Mayrhauser, and A.M. Vans. Program understanding behavior during debugging of large scale software. In *Proceedings of the 7th Workshop on the Empirical Studies of Programmer*, 1997.

[Mirghasemi2011] S. Mirghasemi, J.J. Barton, and C. Petitpierre. Debugging by lastChange. Technical Report. EPFL-REPORT-164250, 2011.

[OCallahan2006] R. O'Callahan. NEfficient Collection And Storage Of Indexed Program Traces. http://robert.ocallahan.org/2006/12/more-about-amber_29.html, December, 2006.

[Pothier2007] G. Pothier, É. Tanter, and J. Piquer. Scalable omniscient debugging. In *22nd annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications(OOPSLA)*, October, 2007.

[Pothier2011] G. Pothier, and É. Tanter. Summarized Trace Indexing and Querying for Scalable Back-in-Time Debugging. In *Proceedings of the 25th European conference on Object-Oriented Programming (ECOOP)*, July, 2011.

[Richards2011] G. Richards, C. Hammer, B. Burg and J. Vitek. The eval that men do. In *Proceedings of the 25th European conference on Object-Oriented Programming (ECOOP)*, July, 2011.

[Richards2010] G. Richards, S. Lebresne, B. Burg and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation(PLDI)*, June, 2010.

[ServerSideJS] Server-Side JavaScript Reference v1.2. http://research.nihonsoft. org/javascript/ServerReferenceJS12.

[Shapiro1982] E.Y. Shapiro. Algorithmic Program Debugging. MIT Press, 1982.

[Stanley2009] T. Stanley, T. Close, M.S. Miller. Causeway: A message-oriented distributed debugger. Technical Report. HPL-2009-78, 2009.

[Tassey2002] G. Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology,* RTI Project Number 7007.011, 2002.

[Thiemann2005] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *Proceedings of European Symposium on Programming (ESOP)*, 2005.

[Tip1995] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.* 1995.

[Vessey1985] I. Vessey. Expertise in debugging computer programs. *International Journal of Man-Machine Studies: A Process Analysis*, 23(5):459-494, 1985.

[Weiser1979] M. Weiser. Program Slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, University of Michigan, Ann Arbor, 1979.

[Yin2011] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering(ESEC/FSE)*, 2011.

[Zaytsev2010] J. Zaytsev. Named function expressions demystified. http:// kangax.github.com/nfe, June, 2010.

[Zeller2002] A. Zeller, and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. IEEE Transactions on Software Engineering 28(2), February 2002, pp. 183-200.

Salman Mirghasemi

CONTACT INFORMATION

School of Computer and Communication Sciences
Éole Polytechnique Fédérale de Lausanne
EPFL IC LTI
Station 14
1015 Lausanne, Switzerland

Phone: +41-21-6936600
Mobile: +41-76-2939899
Fax: +41-21-6936600
E-mail: salman.mirghasemi@epfl.ch
WWW: google salmanmirghasemi

SUMMARY

I have been extensively involved in design and development of software systems in the last 11 years. I am particularly experienced at Java technologies including both server side (Java EE) and client side. During my PhD, I focused on improving software quality by enhancing testing and debugging approaches. The result of my work has been received very well by the software community and supported by different software vendors such as Google, Mozilla and IBM.

EDUCATION

Éole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

Ph.D., School of Computer and Communication Sciences, April 2008 - present

- Thesis Topic: *Querypoint Debugging*
- Advisers: Dr. John J. Barton, Professor Claude Petitpierre
- Area of Study: Software Engineering, Testing and Debugging

Sharif University of Technology, Tehran, Iran

M.S., Computer Science, January 2007

- Thesis Topic: *Employing Semantic Web in Software Engineering*
- Adviser: Professor Hassan Abolhassani
- Area of Study: Software Engineering

B.S., Applied Mathematics, June 2004

WORK EXPERIENCE

IBM, Almaden Research Center, San Jose, California, USA
Intern, June 2010, September 2010
HyperOffice, Rockville, Maryland 20852, USA
Senior Developer, April 2007 - March 2008
Sepidan System Co.,
Senior Developer, April 2006 - March 2007
RoshdAfzar Pegah Co.,
Technical Director, April 2002 - March 2006
Touchtone Corporation, Costa Mesa, CA 92626-4624, USA
Developer, July 2000 - March 2002

TECHNICAL SKILLS

Expert Java programmer, deeply familiar with most Java SE and Java EE APIs, including Servlets, Java Server Pages (JSP), Java Server Faces, JDBC, EJB, JDO, Mail; Experienced user of development tools such as IntelliJ IDEA, Eclipse, AndroMDA, JUnit, Ant, Maven, SVN, Hibernate, Spring

REFERENCES AVAILABLE TO CONTACT

Dr. John J. Barton (e-mail: johnjbarton@google.com)
Professor Claude Petitpierre (e-mail: claude.petitpierre@epfl.ch; phone:+41-21-6932650)