

From Software Failure to Explanation

Sebastian Jeremias Röbler

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes
Saarbrücken, 2013

Day of Defense
Dean
Head of the Examination Board
Members of the Examination Board

July 12, 2013
Prof. Dr. Mark Groves
Prof. Dr. Reinhard Wilhelm
Prof. Dr. Andreas Zeller,
Prof. Dr. Christian Hammer
Dr. Gordon Fraser
Dr. Alessandra Gorla

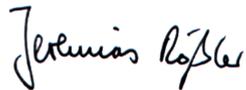
Eidesstattliche Versicherung

nach Promotionsordnung vom 19. Juni 2007, Anlage 1

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. die aus anderen Quellen oder indirekt übernommen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlandung eines akademischen Grades vorgelegt.

Saarbrücken, August 20, 2013



Jeremias Rößler

Abstract

“Why does my program crash?”—This ever recurring question drives the developer both when trying to reconstruct a failure that happened in the field and during the analysis and debugging of the test case that captures the failure.

This is the question this thesis attempts to answer. For that I will present two approaches which, when combined, start off with only a dump of the memory at the moment of the crash (a core dump) and eventually give a full explanation of the failure in terms of the important runtime features of the program such as critical branches, state predicates or any other execution aspect that is deemed helpful for understanding the underlying problem.

The first approach (called RECORE) takes a core dump of a crash and by means of search-based test case generation comes up with a small, self-contained and easy to understand unit test that is similar to the test as it is attached to a bug report and reproduces the failure. This test case can serve as a starting point for analysis and manual debugging. Our evaluation shows that in five out of seven real cases, the resulting test captures the essence of the failure.

But this failing test case can also serve as the starting point for the second approach (called BUGEX). BUGEX is a universal debugging framework that applies the scientific method and can be implemented for arbitrary runtime features (called facts). First it observes those facts during the execution of the failing test case. Using state-of-the-art statistical debugging, these facts are then correlated to the failure, forming a hypothesis. Then it performs experiments: it generates additional executions to challenge these facts and from these additional observations refines the hypothesis. The result is a correlation of critical execution aspects to the failure with unprecedented accuracy and instantaneously point the developer to the problem. This general debugging framework can be implemented for any runtime aspects; for evaluation purposes I implemented it for branches and state predicates. The evaluation shows that in six out of seven real cases, the resulting facts pinpoint the failure.

Both approaches are independent from one another and each automates a tedious and error prone task. When being combined, they automate a large part of the debugging process, where the remaining manual task—fixing the defect—can never be fully automated.

Zusammenfassung

“Warum stürzt mein Programm ab?” – Diese ewig wiederkehrende Frage beschäftigt den Entwickler, sowohl beim Versuch den Fehler so zu rekonstruieren wie er beim Benutzer auftrat, als auch bei der Analyse und beim Debuggen des automatisierten Testfalles der den Fehler auslöst.

Und dies ist auch die Frage, die diese Thesis zu beantworten versucht. Dazu präsentiere ich zwei Ansätze, die, wenn man sie kombiniert, als Eingabe lediglich einen Speicherabzug (“core dump”) im Augenblick des Absturzes haben, und als Endergebnis eine Erklärung des Absturzes in Form von wichtigen Ausführungseigenschaften des Programmes liefert (wie z.B. Zweige, Zustandsprädikate oder jedes andere Merkmal der Programmausführung das für das Fehlerverständnis hilfreich sein könnte).

Der erste Ansatz (namens RECORE) nimmt einen Speicherabzug, der beim Absturz erstellt wurde, und generiert mittels suchbasierter Testfallerzeugung einen kleinen, leicht verständlichen und in sich abgeschlossenen Testfall, der denen die den Fehlerberichten (“bug reports”) beigefügt sind ähnelt und den Fehler reproduziert. Dieser Testfall kann als Ausgangspunkt der Analyse und zum manuellem Debuggen dienen. Unsere Evaluation zeigt, dass in fünf von sieben Fällen der erzeugte Testfall den Absturz erfolgreich nachstellt.

Dieser fehlschlagende Testfall kann aber auch als Ausgangspunkt für den zweiten Ansatz (namens BUGEX) dienen. BUGEX ist ein universelles Rahmenwerk, das die wissenschaftliche Methode verwendet und für beliebige Ausführungsmerkmale des Programmes implementiert werden kann. Zuerst wird der fehlschlagende Testfall bezüglich dieser Merkmale beobachtet, d.h. die Merkmale werden aufgezeichnet. Dann werden aktuelle Methoden des Statistischen Debugging verwendet, um die Merkmale mit dem Testfall zu korrelieren, also um eine Hypothese zu bilden. Anschließend werden Experimente ausgeführt: BUGEX generiert zusätzliche Programmausführungen um diese Korrelation zu prüfen und die Hypothese zu verfeinern. Das Ergebnis ist eine Korrelation zwischen kritischen Ausführungseigenschaften und dem Fehlschlagen des Programmes mit beispielloser Genauigkeit. Die entsprechenden Merkmale zeigen dem Entwickler unmittelbar das Problem auf. Dieses allgemeine Rahmenwerk kann für beliebige Ausführungsmerkmale implementiert werden. Zu Evaluationszwecken habe ich es für Programmzweige und Zustandsprädikate implementiert. Die Evaluation zeigt, dass in sechs von sieben realen Fällen die resultierenden Merkmale den Fehler genau bestimmen.

Beide Ansätze sind unabhängig von einander und jeder automatisiert eine mühsame und fehleranfällige Aufgabe. Wenn man sie kombiniert automatisieren sie einen Großteil des Debugging Prozesses. Die verbleibende manuelle Aufgabe – den zu Fehler beheben – kann nie vollständig automatisiert werden.

Acknowledgments

First and foremost, I want to thank my adviser Andreas Zeller for his patience and support over the years and for the many things I learned from him.

I want to thank my coauthors and colleagues Gordon Fraser, Alessandro Orso, Cristian Zamfir and George Candea for their effort and insights, and for a productive cooperation.

I also want to thank my colleagues at the chair of Software Engineering: Yana Mileva, Valentin Dallmeier, Martin Burger, Kim Herzig, David Schuler, Clemens Hammacher, Kevin Streit, Juan Pablo Galeotti, Alessandra Gorla, Florian Groß, Konrad Jamrozik, Matthias Schur, Sascha Just, Eva May, Matthias Höschele and Andrey Tarasevich. I really enjoyed the friendly and open atmosphere, fruitful discussions, the retreats at Dagstuhl and the attendance of various conferences and workshops.

I am indebted to Kim Herzig, Sascha Just, Sebastian Hafner, Andreas Rau and Curd Becker for keeping the infrastructure at the chair up and running. I am also indebted to Gabriele Reibold, for running the office and helping me with all the many different applications. Thank you.

I also would like to thank my family: My mother Isolde Vongerichten, my father Karl-Heinz Rößler, my brothers Jonathan and Joshua, and my parents-in-law Hildegard and Herbert Schlindwein. And last but not least I want to thank Kerstin (my wife) for ever reliable support and loving help, and Henry and Nora (my two kids) for giving me strength and confidence.

Contents

1	Introduction	1
1.1	An Ongoing Example	4
1.2	When are Two Failures Equivalent?	4
1.3	Reproducing a Failure	6
1.4	Understanding a Failure	7
1.5	Publications	9
1.6	Thesis Structure	9
2	Background	11
2.1	Automated Test Case Generation	11
2.2	Genetic Algorithms	13
2.3	Record and Replay	15
2.4	Execution Synthesis	17
2.5	Defect Localization	17
2.5.1	Statistical Debugging	18
2.5.2	Experimental Debugging	20
2.6	Program Understanding	20
2.7	User Studies	22
2.8	Bug Benchmarks	23
3	From Failure to Test	25
3.1	An Example	26
3.2	Implementation Details	29
3.2.1	Collecting Crash Data	30
3.2.2	Fitness Function	31
3.2.3	Seeding Primitive Values	35
3.2.4	Limitations	36
3.3	Case Studies	37
3.3.1	JODATIME Brazilian Date Bug	38
3.3.2	Vending Machine Bug	39
3.3.3	Commons Math Sparse Iterator Bug	39
3.3.4	JODATIME Western Hemisphere Bug	40
3.3.5	JODATIME Parse French Date Bug	40
3.3.6	Commons Codec Base64 Decoder Bug	41
3.3.7	Commons Codec Base64 Lookup Bug	42
3.3.8	Threats to Validity	42
3.3.9	Summary	43

CONTENTS

4 From Test to Explanation	45
4.1 The First Approach	45
4.1.1 Intuition Behind the Approach	46
4.1.2 Implementation Details	46
4.1.3 Simplifying Constraints	49
4.1.4 Limitations	50
4.2 The BugEx approach	53
4.2.1 Explanatory Facts	54
4.2.2 The Approach in Detail	56
4.2.3 Implementation	62
4.2.4 Evaluation	64
4.2.5 Quantitative Results	66
4.2.6 Qualitative Results	68
4.2.7 Limitations	76
4.3 User Study	76
4.3.1 Research Questions and Hypotheses	77
4.3.2 Design of the User Study	78
4.3.3 How the Study Failed	83
4.3.4 Conclusions	84
5 Conclusion and Future Work	85
5.1 Conclusion	85
5.2 Future Work	87
5.2.1 Large-Scale Evaluation	87
5.2.2 Future Work on RECORE	94
5.2.3 Future Work on BUGEX	95
Bibliography	97

List of Figures

1.1	Request to send an automatically created crash report	2
1.2	Bug report for JODATIME bug 2487417	4
1.3	Test case for JODATIME bug 2487417	4
1.4	Stack trace of JODATIME bug 2487417	5
2.1	Example of a state space landscape	14
3.1	Initial test case as generated by RECORE	27
3.2	Test case as generated by RECORE after feedback	27
3.3	Mutated test case as generated by RECORE	28
3.4	Final test case generated by RECORE	28
3.5	Overview how RECORE works	29
3.6	The try-catch that is put around each method	30
3.7	Methods added to the <code>java.lang.Throwable</code>	31
3.8	Code added to the client's main class	31
3.9	Date in number of seconds	39
3.10	Date in calendar form	39
3.11	Test generated by RECORE for the Vending Machine bug	39
3.12	Test generated by RECORE for the Sparse Iterator Bug	40
3.13	Test generated by RECORE for the Parse French Date Bug	41
3.14	Description of the Base64 Decoder bug as found in the bug report	42
3.15	Test case generated by RECORE for the Base64 Decoder bug	42
3.16	Test case generated by RECORE for the Base64 Lookup bug	42
4.1	Failure conditions for the JODATIME Brazilian example	46
4.2	Intuitive view of the approach	47
4.3	Simplified pseudocode that depicts the approach	48
4.4	Simple example to further illustrate the approach	49
4.5	Example of a predicate that contains an OR condition	51
4.6	Intuitive view of how the generated execution paths relate	52
4.7	Conditions necessary to trigger the Brazilian example	54
4.8	Date of the next daylight savings time transition	55
4.9	Invariants violated by the failing test of the Brazilian example	55
4.10	Definition-usage pair for the Brazilian example	55
4.11	Branch correlated to the Brazilian example	56
4.12	Interesting loop of the Brazilian example	56
4.13	BUGEX in a nutshell	57
4.14	Simplified pseudocode for BUGEX	57

LIST OF FIGURES

4.15	Example code that can throw a <code>NullPointerException</code>	58
4.16	Explicit <code>null</code> check in the code	58
4.17	Number of branches remaining over time	67
4.18	Branch returned by BUGEX for the Vending Machine example	69
4.19	Test to trigger the Sparse Iterator Bug	70
4.20	Bug report of the Base64 Decoder bug [21]	71
4.21	Code excerpt for the Base64 Decoder bug	71
4.22	Report of the Western Hemisphere bug	72
4.23	Survey on the last committed revision	79
4.24	Additional question on the last committed revision	80
4.25	Question on the fix committed in the last revision	80
4.26	Additional question on the fix committed in the last revision	81
4.27	Optional questions on the fix committed in the last revision	81
4.28	Survey on the last diagnosis the student received	81
5.1	A simplified scheme of TRUEBUG	88
5.2	How TRUEBUG extracts minimal defect and fix	89
5.3	Percentage of mined projects compatible with TRUEBUG	90
5.4	Distribution of unique build systems per project	91
5.5	Distribution of integrated development environments	92

List of Tables

3.1	RECORE evaluation subjects	37
3.2	Summary of results	38
4.1	BUGEX evaluation subjects	65
4.2	Number of facts reported by BUGEX	66
4.3	Number of branches reported by statistical debugging	68

Chapter 1

Introduction

If debugging is the process of removing bugs,
then programming must be the process of putting them in.

— Edsger W. Dijkstra

Programs are made by humans and humans make mistakes. A mistake in the code of a program is called a *defect* or less formal a “bug”. The process of removing or fixing bugs is called *debugging*. Debugging is a tedious task that accounts for about 50% of the effort spent on developing and maintaining a piece of software [65]. Thus any technique that can enhance or automate parts of that process is of high practical value and helps to cut both time and cost in the overall development process.

Many defects are not found by reviewing the source code, but because of the observable erroneous program behavior they cause, which is called a *failure*. Some failures present themselves as *crashes*, that is they cause the program to terminate abruptly or to be terminated by the runtime system; often discarding valuable user input (which can amount to hours of work) and leaving resources (e.g., files and connections) in inconsistent state.

Every nontrivial program operates on some sort of internal data, called the *execution state*. Often, a defect corrupts that execution state, causing an *infection*. A failure only can occur if the program subsequently operates on the corrupted data. This leads to a distance between the time the defect is executed and the state is infected, and the time the failure is observable. And it also means that, even if the defect is triggered, it does not always need to show.

All defects are differences between the expected and the actual program behavior. They can only be identified *relative* to an explicit or implicit specification. Generally, defects are neither self-evident nor independent; by examining the program in isolation, defects cannot be detected. Defects could also be called *surprises* as in “without specification, there are no bugs—only surprises” (Brian Kernighan).

However, there is an implicit and absolute specification that applies to *all* programs. A program should be able to respond to all possible conditions. If it is unable to proceed because it is in an erroneous state, it should terminate gracefully with an appropriate error message, and, if possible, save user input and leave used resources in a consistent state. Under no circumstances is it acceptable for a program to crash. A defect relative to this absolute specification, i.e., a defect that results in a crash of the program, is a self-evident defect. Of course whether a defect results in a crash depends on the ability of the runtime system to detect certain erroneous states (e.g., division by zero, illegal memory access, ...). So the same defect may on one runtime system result in a crash and on a different runtime system—or even a different configuration of the same runtime system (e.g., if assertions are disabled)—go undetected.

Software specification is a vast field in its own right, with which this thesis is not concerned. Thus in the following, only self-evident defects are considered; and every time the term failure is used, it denotes a crash. However, since such defects are only a specialization of relative defects, it should be straightforward to adapt the approaches if a general specification of the program is available.

Debugging is a formalizable process consisting of the following subtasks:

Registering the failure In order to be able to manage the failure, it must first be registered. This can happen automatically, e.g., if a program crashes on an end user’s machine, the user is often requested to send an automatically created crash report to the program manufacturer via the push of a button. Usually, this includes a memory dump or *core dump*¹. Systems in which the failure can be registered and managed (e.g., documented, assigned and tracked) are called bug tracking systems.



Figure 1.1: Request to send an automatically created crash report

Verifying and reproducing the failure Before the failure can be acted on, its existence must first be verified by reproducing it (see Section 1.3). This manual reproduction is then turned into an automated reproduction by means of a simplified automated test case. Having an automated test case is important, because if the failure cannot be replayed at will, it cannot be verified that the fix actually causes the failure to disappear. An automated test case also helps the developer to investigate the failure in the next step.

¹ The name stems from magnetic core memory, the principal form of random access memory from the 1950s to the 1970s.

Understanding the failure For the developer to make an informed decision of how they want to fix the defect, they first have to understand the causation chain which eventually leads to the failure. And since they really start with the failure and not with the defect, they have to backtrack the infection that causes the failure to the defect that causes the infection.

Correcting the defect In general, any given defect has unlimited ways of being fixed, including a complete rewrite of the program from scratch. Any change that breaks the causation chain that eventually triggers the failure is a valid fix. *Which* fix the developer settles for is a complex decision, depending on many different factors. And since the fix defines the location of the defect, it can never be fully automatically localized. However, once the fix is applied, the automated test case becomes important to verify that the fix actually made the failure disappear.

Automated debugging is the scientific field that tries to automate parts of the debugging process. In this thesis, two approaches are presented which represent two major contributions to the field of automated debugging:

Generating a Test Case The task to create an automated test case that replays the failure is both important and can be very difficult. Which is the reason why it is common for open source projects to *require* an automated self-contained test case for the very act of registering a bug. The difficulty stems from the fact that, in order to create that test case, one usually has to already have an idea of how the failure comes into existence. And to get this idea is very difficult if one cannot replay the failure at will or at least has some knowledge about the circumstances in which the failure occurred.

RECORE ([Chapter 3](#)) takes a core dump as often supplied from a user and automatically generates an automated and self-contained test case that triggers the failure, effectively automating the complete task. It does so by generating random test cases and using a genetic algorithm to evolve the test cases towards triggering the same failure and producing a similar core dump. To greatly speed up the process, it reuses values found in the original core dump.

Explaining a Failure As already pointed out, the task to localize the defect can never be fully automated, because in general, for each failure, there are unlimited possibilities to fix it. However, a tool can help a developer to *understand* how a failure comes to be, by highlighting differences between runs that lead to the failure and similar runs that will not fail, and by pointing to critical aspects and characteristics of how the data interacts with different parts of the code.

BUGEX ([Section 4.2](#)) implements a general debugging framework, by employing the scientific method. It *observes* different runtime aspects (called *facts*) of the failing run. It then uses state-of-the-art statistical debugging methods to rank these facts, creating a *hypothesis* about which facts correlate with the failure. Then it performs *experiments*: it generates additional executions to challenge these facts and from these additional observations refines the hypothesis (i.e., the correlation of the facts to the failure). The result is a correlation of facts to the failure with unprecedented precision. The whole framework can be applied to any runtime information one deems of interest; to demonstrate its power, it was implemented for branches and state predicates.

1.1 An Ongoing Example

As an ongoing example, that will be used throughout this thesis, consider the failure that is reproduced by the JAVA test case shown in Figure 1.3. This test case stems from the bug report 2487417 of the JODATIME date and time library [62]. It produces a certain date (October 18, 2009) and transforms this date to a time interval that represents the 24 hours of that date in the Brazilian time zone. Figure 1.2 shows the description of the failure as found in the bug report.

I am using `LocalDate.toInterval()` to compute ‘all day’ intervals as specified: “Converts this object to an Interval representing the whole day.”

This works perfectly with one exception: on daylight cut days in a Brazilian timezone. `LocalDate.toInterval()` specification is “Converts this object to an Interval representing the whole day”. While “00:00:00” does not exist, the all-day interval does exist I think.

Figure 1.2: Bug report for JODATIME bug 2487417

```

1 public class JodaTest extends TestCase {
2     public void testBrazil() {
3         DateTimeZone dtz =
4             DateTimeZone.forID("America/Sao_Paulo");
5         LocalDate date = new LocalDate(2009, 10, 18);
6         Interval interval = date.toInterval(dtz);
7     }
8 }

```

Figure 1.3: Test case that reproduces JODATIME bug 2487417: `toInterval()` fails when processing October 18, 2009, Brazil time

The reason why this test fails is that October 18, 2009 is the last day of daylight savings time (DST) in Brazil, a condition that is improperly handled by JODATIME due to an inconsistent mapping between UTC time (the internal intermediate format) and local time. It basically boils down to the fact that JODATIME tries to create a time interval that starts at midnight, but an internal consistency check recognizes that on that date, there was no midnight—and raises an `IllegalArgumentException` with the message and stack trace as shown in Figure 1.4.

1.2 When are Two Failures Equivalent?

An important question when dealing with failures is: When are two failures identical or equivalent? The obvious answer is: when they are triggered by the same defect. But as pointed out earlier, the defect in turn is defined by its fix. And since there are infinitely many fixes for any given failure, so are there

```

java.lang.IllegalArgumentException:
  Illegal instant due to time zone offset transition:
  2009-10-18T03:00:00.000
at org.joda.time.chrono.ZonedChronology.localToUTC
  (ZonedChronology.java:143)
at org.joda.time.chrono.ZonedChronology.getDateTimeMillis
  (ZonedChronology.java:119)
at org.joda.time....AssembledChronology.getDateTimeMillis
  (AssembledChronology.java:133)
at org.joda.time.base.BaseDateTime.<init>
  (BaseDateTime.java:254)
at org.joda.time.DateMidnight.<init>
  (DateMidnight.java:268)
at org.joda.time.LocalDate.toDateMidnight
  (LocalDate.java:740)
at org.joda.time.LocalDate.toInterval
  (LocalDate.java:847)

```

Figure 1.4: Stack trace of JODATIME bug 2487417

infinitely many equally valid identifications of the defect. In addition to that, the defect is an intermediate result of the debugging task. And since humans make mistakes also when debugging, we need an equivalent failure to prove that the defect actually was correctly identified and fixed. So how can one know whether two failures are equivalent before debugging them and finding the defect that causes them, in spite of the circular dependency?

Ultimately, there is no way to decide this. In theory, the only time one can be sure two failures are equivalent is if *every aspect of the program executions* are completely equivalent. Since a program is but a mathematical mapping of inputs to outputs, all it takes is to record the inputs to reproduce the output. However, there are many aspects of input [126]:

User Interaction This is what first comes to mind when talking about inputs: the input from the user in its many possible forms. And this is also the easiest input to track and store. However, if stored for debugging, this may raise privacy issues.

Data In many different forms, data is stored in files and databases. This input is still relatively easy to track and store. However, in addition to privacy issues, the sheer amount of data can be a problem.

Time Reproducing the exact time a program was started is already quite a challenge, but still feasible.

Randomness For some programs (e.g. games and cryptographic applications) nondeterminism is part of the design. In that case it suffices to capture the seed of the pseudorandom number generator. But there are many more ways how randomness is introduced into a program. For instance, hash values in JAVA are calculated from the memory addresses that the objects in question

reside in—which depend on so many factors that they are next to impossible to faithfully reproduce. And this is just one example of the many ways unintended randomness finds its way into programs.

Schedules Execution schedules of multiple threads in a program and multiple instances (processes) of the same program are also hard to reproduce, as they are defined by the runtime or operating system and are usually outside of the control or even the perception of the developer.

Operation Environment This includes the operating system, the specific architecture, the file structure, other programs installed on the system, and every third party library and system function that is used by the program. In quintessence it is already hard to create the exact same environment on a different computer of the same model and impossible to do this on a computer of a different model.

Physics If one was able to faithfully reproduce all other aspects of an execution, this is the last barrier, where finally practical reality forces its way into the theoretical virtuality that a computer program constitutes. Simple physical deviations may cause deviations and errors in the program and are truly impossible to exactly reproduce.

So in practice, it is simply infeasible to reproduce every aspect of a program execution. And in addition to that, it is of no practical value. Because not the failure is in the center of the focus, but the defect. So one should settle for a heuristic that approximates the fact that two failures are equivalent if they are caused by the same defect. An easy to use heuristic that showed to work reasonably well and is therefore widely used in practice is to treat two failures as equivalent if the same exception was raised from the same location in the program (as indicated by line number of source code).

So for the failure with the example stack trace as given in [Figure 1.4](#), this would mean any other failure is equivalent to it, if the raised exception is a `java.lang.IllegalArgumentException` with the same message (“Illegal instant due to time zone offset transition: 2009-10-18T03:00:00.000”) and if the location it was raised at is the `org.joda.time.chrono.ZonedChronology` class and the line number is 143 (which is in the `localToUTC()` method). Note that the rest of the stack trace is not considered in this heuristic, as the same defect might be triggered with different call stacks.

1.3 Reproducing a Failure

When a program fails in the field, the developer who wants to debug the issue must first be able to *reproduce* the problem by raising an equivalent failure (as discussed in [Section 1.2](#)). Being able to reproduce a failure locally is important for diagnostic purposes, because developers can then inspect arbitrary aspects of the failing execution to locate the failure cause. If the problem cannot be reproduced, though, one can never know whether the fix is successful: Only if

the failure goes away after the fix has been applied does the developer know that she effectively found the failure cause.

The stack trace, which is the default output in JAVA in case of an error, suffices in practice to decide if two failures are equivalent, as it contains both the exception and the location in the code where the exception was raised (as discussed in [Section 1.2](#)). However, the stack trace does *not* suffice to trigger the same failure, as it does not contain any information of what input or method sequence lead to reaching the particular location in the code.

To be able to reproduce the failure, one needs to record (at least some) input of the program. But as was already shown in [Section 1.2](#), the problem in practice is that recording inputs incurs time and memory overhead, and neither developers nor users may be willing to spend resources on a diagnostic feature that (in the best of cases) will never be used.

Once a failure *has* occurred, though, such diagnostic overhead no longer matters. A common practice on modern operating systems is to produce a *core dump*—a persistent snapshot of the stack and heap memory at the moment of the failure (see [Figure 1.1](#)). Such a core dump can be easily transferred to a developer, who can then load it into an interactive debugger and inspect it post-mortem to understand how the failure came to be.

Such a core dump, however, just reports the final state, not the history. Assuming one finds an illegal value in some variable: How did that value come to be? The input that caused the failure may long have been overwritten by later input: What was it that caused the problem in the first place? With a core dump, you are a detective faced with a corpse—and you have to reconstruct all the events that lead to it.

So for the failure as given in [Section 1.1](#), this means that the core dump contains the offending time zone offset transition. Other values, such as the `DateTimeZoneID` are stored in the respective objects as local variables and are also part of the core dump. However, intermediate values that are needed to create the infected state, such as the concrete date or a certain method sequence are likely to be lost. So with just a stack trace (as given in [Figure 1.4](#)) a developer will have to invest some effort to create a simple test case as shown in [Figure 1.3](#)—if they succeed at all.

1.4 Understanding a Failure

“Why does my program crash?”—This ever recurring question of software debugging drives the developer during the analysis of the failure. As discussed earlier, a program can be fixed in infinite many different ways. Several heuristics have been shown to reasonably well identify simple defects similar to how developers do so. However, complex defects are impossible to automatically identify; this can only be left to human judgment.

But what we can do is empower the developer to make an informed decision, by helping them understand the failure. Manual debugging is driven by experiments, test runs that narrow down failure causes by systematically confirming

or excluding individual factors. To fully comprehend a failure, one may need to consider many different aspects such as the range of the input parameters, the program's structure and its runtime behavior. These explain different aspects of the failure and help the developer understand and eventually fix the underlying defect.

For instance, in order to understand the failure that is described in [Section 1.1](#), the developer might ask questions such as “What is special about Brazil as a time zone in regard to that particular date?” and “Why does this make my program crash?” If a developer wants to write a fix for the underlying defect or even only assess it, they have to investigate the code and do some manual debugging to understand how the failure comes into existence. And since this three lines long test case invokes 367 methods and executes 1,465 lines of code (ignoring JAVA system libraries), they might have a hard time doing so.

The state of the art in automated debugging focuses on *statistical* approaches to defect localization [2, 44, 64, 75, 76, 77, 78]. These techniques effectively *rank* the lines of code according to their likelihood of containing the defect. Given a sufficiently high number of executions, one can expect a statistical approach to narrow down the search space to less than 5% of the code. While these numbers are impressive, the result is still not necessarily helpful for the programmer, as 5% of the code may still encompass thousands of lines of code. The evaluation of those approaches usually assumes an “ideal user” [99], who immediately recognizes the defect on sight. This convenient assumption allows for an easy and objective evaluation. But this assumption is also very questionable [94]: After some time has passed, developers usually do not understand their own code, let alone someone else's. If only the problematic lines of code are presented without explanation or context, *developers will not recognize the defect*. Additionally, as recent research has shown, developers are unwilling to proceed through long lists of unrelated suspicious locations, with small chances of spotting the defect [94]. And even if the code is almost correct and the fix is simple, without understanding the failure *developers cannot write the fix*.

And additional to that, statistical debugging relies on the strong assumption that test suites include large numbers of test cases, which is unfortunately rarely the case in practice. And even if so, these test cases are usually optimized for coverage, which means that they try to execute as much of the code as possible at least once. Statistical debugging on the other hand would benefit from having a small part of the code being executed extensively, to generate as much relevant and diverse input to the calculation of the correlation as possible

In contrast, little effort has been spent to help developers *understand* a failure. And for an obvious reason: the evaluation depends on human feedback and thus is much more laborious and error prone. Yet this is a much better indicator of the overall goal of developer productivity.

The fix for the example from [Section 1.1](#) involves an API change with deprecation of several methods and classes, and the development of new classes and methods to replace them. For such cases most approaches to defect localization do not even fail well—they are not designed to detect when the fix involves more than a single line of code.

1.5 Publications

This dissertation builds on the following papers (in reverse chronological order):

Reconstructing Core Dumps - ICST 2013

by **Jeremias Rößler**, Andreas Zeller, Gordon Fraser, Cristian Zamfir and George Candea.

ICST 2013: Proceedings of the 2013 International Conference on Software Testing, Verification and Validation, March 2013

Isolating Failure Causes through Test Case Generation - ISSTA 2012

by **Jeremias Rößler**, Gordon Fraser, Andreas Zeller, Alessandro Orso.

ISSTA 2012: Proceedings of the 2012 International Symposium on Software Testing and Analysis, July 2012.

How Helpful Are Automated Debugging Tools? - USER 2012

by **Jeremias Rößler**.

USER 2012: Proceedings of the first Workshop on User evaluation for Software Engineering Researchers, June 2012.

Understanding Failures Through Facts - ESEC/FSE 2011

by **Jeremias Rößler**.

ESEC/FSE 2011: Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 2011.

When does my program fail? - CSTVA 2011

by **Jeremias Rößler**, Alessandro Orso, Andreas Zeller.

CSTVA 2011: Proceedings of the 3rd Workshop on Constraints in Software Testing, Verification, and Analysis, March 2011.

All experiments and experimental setups presented were designed and conducted by the author of this thesis. However, the author was supervised and reused existing tools and techniques. Therefore, most of the papers that this dissertation builds on had additional authors. Referring to the team that contributed to the work of which this thesis is part of, the personal pronoun “we” rather than “I” is used throughout the thesis.

1.6 Thesis Structure

The remainder of this thesis is structured as follows: In [Chapter 2](#), we give a broad overview over similar and related approaches and introduce some basic methods and techniques that our approach builds upon. In [Chapter 3](#) we introduce RECORE, a technique that, starting only from a core dump, employs a genetic algorithm to generate a unit test that reconstructs the failure. In [Chapter 4](#) we present our efforts to help a developer understand the failure at hand. First we illustrate an initial attempt to solve the problem ([Section 4.1](#)), and then proceed to BUGEX ([Section 4.2](#)), a refined technique that can be understood as

a universal framework to explain a failure in terms of arbitrary runtime aspects. In [Section 4.3.2](#) we give a short outline on how started to evaluate BUGEX with a user study. We finish the thesis with conclusions and future work ([Chapter 5](#)), which contains a scheme for a large scale evaluation and a general benchmark for debugging tools.

It should be noted, that the work the individual chapters base upon, was performed in different chronological order than the order of the chapters suggests. But we preferred to order the chapters according to the debugging process, rather than to the mostly arbitrary sequence in which the research was conducted. As can be seen in [Section 1.5](#), we started with trying to help developers understand failures ([Section 4.1](#)). To overcome the limitations of our first approach, we started all over again and eventually came up with a different approach we called BUGEX ([Section 4.2](#)). In order to correctly evaluate BUGEX, we prepared a user study ([Section 4.3](#)) and a large scale automated evaluation ([Section 5.2.1](#)).

Chapter 2

Background

If I have seen further it is by standing on the shoulders of giants.

— Isaac Newton

This work builds on understanding and reusing research and work of preceding thinkers and really would not have been possible without. We rely on the state-of-the-art of many existing fields and specialities and employ different existing techniques by combining them in new ways or applying them to new areas.

In the following, we will illustrate some basic state-of-the-art techniques that this work is based on, and present some similar and related leading edge approaches, that have yet to be challenged and refined by a broader body of research.

2.1 Automated Test Case Generation

Both RECORE and BUGEX heavily rely on execution generation techniques. In practice, test cases usually are executions of a program that assert some sort of result (i.e., check whether the program conforms to a specification). However, if no specification is available, it is difficult to classify the execution result of generated test cases as failing or passing. This is the well-known *oracle problem* [115].

One possibility is to generate test cases without explicit oracles, partly relying on the runtime system to detect more severe problems. When doing so, there is no difference between test cases and program executions. Therefore, in the following, we will use the term *test case generation* synonymous to *execution generation*.

Automated test case generation has made significant progress in the recent past, making it possible to derive test inputs that in many cases can reach

large parts of the code. A straightforward and often used approach is to simply generate these test inputs randomly and thus is called *random testing* (e.g., Randoop [92]). This is computationally cheap and large numbers of tests can be produced in a short time. As random tests are unlikely to reach parts of the code that require the traversal of complex predicates, more systematic approaches have been presented.

Symbolic execution [67] is a technique that analyses the program by tracking symbolic rather than actual values and explores the complete execution space. However, for most real-life programs the execution space is infinite and its complete exploration infeasible. Symbolic execution suffers from the problem of state explosion [111].

Dynamic symbolic execution (DSE, e.g., DART [38], Klee [15]) tries to circumvent this problem by collecting, during a concrete execution e , path constraints that encode the subdomain of inputs that will follow the same program path as e . By systematically negating individual clauses in these path conditions, it iteratively expands to additional executions, adding more information and increasing the precision in the process. One can theoretically explore all possible paths, however, this also allows to abort at any stage and still get some results. But since the execution space is only partly explored, DSE is not complete. For instance it generates a set of constraints on the input that could possibly be further reduced; thus it does not produce the weakest precondition [25]. In return the approach scales much better and is applicable to programs of any size. DSE has been integrated in popular tools, such as Pex [110] and Sage [39].

An alternative approach is to cast the test generation problem as a search problem (*search-based testing*, SBST) and use efficient meta-heuristic search algorithms to produce solutions, that is, test cases [86]. This has been shown to be particularly useful if individual test cases are not just primitive input values to a function, but rather complex sequences of method calls. One main advantage of search-based testing is that it is very easy to adapt it to new testing objectives (such as reconstructing core dumps), and tests can be optimized towards different functional and non-functional properties [86].

One of the most popular search algorithms applied in SBST is a Genetic Algorithm (see Section 2.2), where a population of candidate solutions is evolved using operators that intend to imitate natural evolutionary processes. A fitness function determines how close a candidate solution is to the optimization target, and the fitter an individual is, the more likely it is used during reproduction, which is based on selection, crossover and mutation. For example, the recent EVOSUITE prototype [32] evolves test suites towards satisfying a coverage criterion using a genetic algorithm and is applicable to any Java library that requires no user input. Alternatively, EVOSUITE optimizes individual test cases towards satisfying an individual goal. A test case is a sequence of statements (e.g., constructor and method calls, primitive value assignments, field assignments, array assignments), and mutation modifies a sequence by deleting, adding, and replacing statements. For more details about the search operators we refer to Fraser and colleagues [34]. A promising avenue of research in this direction considers combinations of the individual techniques [52, 82].

So far, there has been surprisingly little work leveraging test case generation for debugging purposes—unless one qualifies delta debugging (see [Section 2.5.2](#)) as a test case generator. Baudry and colleagues [11] suggest to improve test suites for defect localization by adding mutated tests; they specifically aim for diversity in defect-diagnosing distinguishing statements. Artzi et al. [7] do not assume the presence of an existing test suite, but rather generate tests from scratch that aim for diversity in the attributes relevant for statistical debugging, such as statements covered, branches taken, or function return values. Both approaches generate *general* test suites to facilitate arbitrary debugging tasks and have been shown to be effective at that. In contrast, BUGEX generates tests to provide a precise diagnosis for a *single given failure* and follows an *experimental* approach that uses feedback from test outcomes to guide test generation. Because BUGEX generates and executes tests for a specific debugging problem, it might overall generate many more tests than these alternative techniques. However, this additional cost is offset by the ability to specifically isolate very precise failure causes.

2.2 Genetic Algorithms

As explained in [Section 2.1](#), both RECORE and BUGEX are based on EVOSUITE to generate test cases. Since EVOSUITE implements a genetic algorithm, we slightly expand on that class of search-based algorithms. These fundamentals are illustrated following Russell and Norvig [105].

In the following, we are concerned with finding the solution to an arbitrary optimization problem. The *search space* of a problem is the solution space as defined by the set of all possible *candidate solutions*. If the search space is too large to be explored systematically (e.g., if it is infinite/continuous), *local search* algorithms were shown to produce “good enough” solutions in reasonable time. They maintain only a single *current state* (rather than e.g. a search *path*) and thus operate with a constant amount of memory, regardless of the size of the search space. In general, local search algorithms move only to neighboring states of the current state (hence the term “local”).

The shape of the search space (called state space *landscape*) is defined by an objective function. Such a state space landscape has a “location” (defined by the state) and an “elevation” or objective value (defined by the value of the state according to the objective function). Depending on whether the problem is cast as an maximization or minimization problem, the aim is either to find the lowest valley (a global minimum) or the highest peak (a global maximum). Since these two problems can easily be converted from one to the other, in the following we will only consider maximization problems. Often, such landscapes also contain local maxima (a peak that is higher than each of its neighboring states, but lower than the global maximum) and plateaux (areas where the objective function is flat). An example of such a state space landscape is given in [Figure 2.1](#). Note that in practice, one often has multiple optimization parameters, rendering the search space as a multidimensional landscape.

A *hill-climbing* search is simply a loop that moves to neighboring states in the direction of increasing value (that is, uphill) and terminates when it reaches

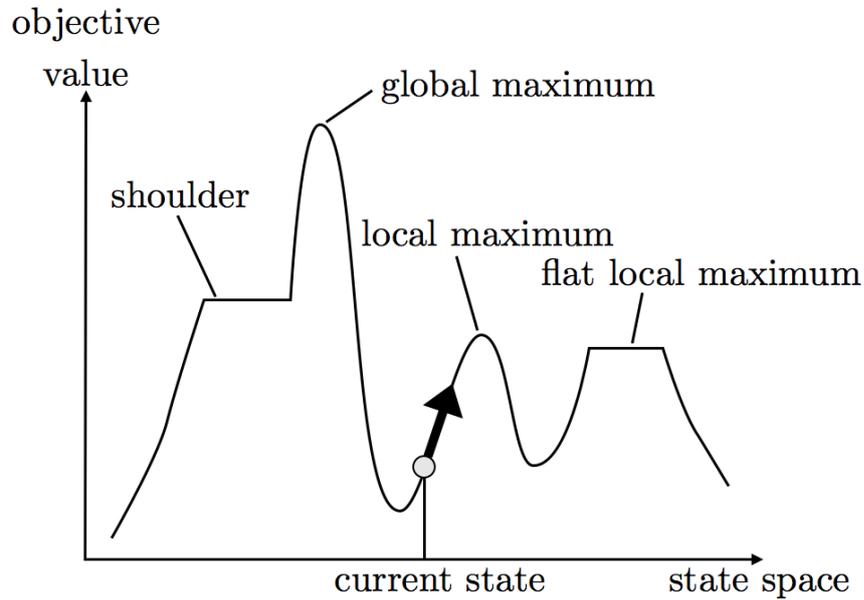


Figure 2.1: Example of a one-dimensional state space landscape as defined by an objective function (following Russell and Norvig [105])

a peak. This simple greedy algorithm often makes rapid initial progress but gets stuck on local maxima (instead of global maxima) or plateaux (where it lacks “guidance” in which direction to proceed).

To circumvent this, *stochastic* hill climbing chooses at random from several possible uphill moves (i.e. if the landscape is multidimensional). In case of many (e.g. thousands) of neighboring states or successors, one can implement stochastic hill climbing as *first-choice* hill climbing by generating successors randomly until one is generated that is better than the current state.

If the successor can also be worse than the current state, this is called random walk. A purely random walk is highly inefficient, but has better chances of not getting stuck at local maxima. *Simulated annealing* (named after the annealing process in metallurgy), also accepts a worse successor by a certain probability that decreases over time. *Random-restart* hill climbing conducts a series of hill-climbing searches, starting from randomly generated initial states.

Instead of a single current state, the *local beam search* algorithm keeps track of k states. At each step, it generates all successors of all k states, but keeps only the k best successors of the complete list. Although this may initially seem like running k random-restart hill-climbing algorithms in parallel, the difference is that it quickly abandons unfruitful searches, concentrating resources on where most progress is being made. Local beam search can suffer from a reduction of diversity among the k states (e.g. if a single state has more than k successor states which are superior to any other successor state), concentrating the states

in a small region of the state space and making it little more than an expensive version of hill climbing. *Stochastic* beam search (analogous to stochastic hill climbing) tries to circumvent this problem by choosing k successors at random with the probability being an increasing function of the objective value.

When calling the successors “*offspring*” and their objective value “*fitness*”, stochastic beam search resembles an evolutionary process of asexual reproduction and natural selection. A *genetic algorithm* or GA makes this a sexual reproduction by combining two parent states or *individuals* instead of modifying a single state. This combination is called *crossover* and highly dependent on the problem domain. For instance, if the individuals are sequences of method calls, a crossover could mean that they are cut at a random crossover point and the two different halves are combined (as is implemented in EVOSUITE).

Depending on the diversity of the parents, crossover can produce individuals that are quite different from either parent. The population is often more diverse in the beginning, so crossover (like simulated annealing) makes the algorithm take large steps in the search space early in the process and smaller steps later when individuals become more similar. Crossover also bears the advantage that it can combine different parts of individuals that evolved independently and perform useful functions, thus raising the level of granularity at which the search operates. Eventually, as in stochastic beam search, all individuals are also modified or *mutated* by a certain probability. The next generation of the population is then chosen from the newly generated ones, according to the objective function or *fitness function*. As was pointed out above, this fitness function defines the search space landscape and thus is highly important for the effectiveness of the algorithm. For instance if the fitness function renders the search landscape flat, the evolutionary algorithm degrades into simple random search.

Of course, genetic algorithms are a crude simplification of the complex real-world evolutionary process. From the many differences, the most important is probably that the natural genes themselves encode parts of the mechanisms of the biological processes (e.g., mate selection, crossover and mutation processes and probabilities, ...) as well as the mechanisms by which the genes are translated into an organism. So in nature, the individuals as well as *the search algorithm itself* are improved.

2.3 Record and Replay

As was detailed in [Section 1.3](#), before a developer can debug a failure, they must first be able to reproduce it on their local machine. Most approaches to reproducing failures rely on *record/replay*: The idea is to *record* events first, and to *replay* them later. This works well for low-bandwidth event sources; recording a user’s input on a GUI, for instance, is unlikely to have a perceptible impact on performance.

As larger amounts of data are recorded, overhead becomes significant. At the system level, record/replay systems such as ReVirt [27], PRES [93] and

ReSpec [73]) and hardware and/or compiler assisted systems (e.g., Capo [89], CoreDet [12]) support efficient recording of multi-core, shared-memory-intensive programs. However, these systems incur high runtime overhead due to recording or require specialized hardware that is not widely available. For data centers, for instance, making up for a 50% throughput drop requires provisioning twice as many machines. The additional storage required for recording also matters: record/replay systems [4] for datacenter applications report significant log sizes even after applying aggressive logging optimizations.

For general-purpose software, record/replay has been mainly explored in the context of *automated debugging*. ADDA [18] records events at the level of C standard library and file operation functions. JINSI [14] records the interaction between individual objects in terms of methods called and objects passed. Such recorded information is great for debugging runs in the lab, as automated debugging tools can manipulate the event stream to isolate failure causes; the JINSI tool, for instance, was evaluated to narrow down the defect search space to 0.22% of the code. The large overhead of such systems, however, limits data recording to the lab.

If one is willing to suffer execution overhead, one can record crucial information required to reproduce the error. The BUGREDUX framework [60] can collect execution data in the field and (like RECORE) use test case generation to reconstruct the failure. In its evaluation, the authors found stack traces collected during execution to provide the greatest benefit compared to the overhead in collecting the data. Fitnex [119] combines elements of both techniques: It is a fitness-based search strategy used to guide the Pex [110] symbolic execution engine toward a certain goal (e.g., an uncovered line of code). The fitness function uses the predicates that appear in branch instructions to estimate how close an execution path is to a particular test target. However, unlike RECORE, Fitnex does not use any of the information in the core dump to evaluate fitness. Likewise, SherLog [120] uses clues from existing application logs to reconstruct the path to a failure using static analysis. Subsequent enhancements of SherLog [122, 121] improve the ability to reproduce failures by using additional recording (and additional runtime overhead).

A little collected information can go a long way. The RECRASH tool [8] records executions by recording parts of the program state at each method entry—namely those objects that are reachable via direct references. When the program crashes, it thus allows the developer to observe a run in several states before the actual crash. Even just recording these references results in a 13%–64% performance overhead. However, it allows the programmer to examine the last few steps before the failure. If the failure-inducing state was created outside of the last stack trace, however, RECRASH will only show that feeding this state caused the crash, but give no indication on how the state came to be.

As stated before, RECORE aims at *zero overhead*, which sets it apart from all such record/replay systems. Like BUGREDUX, its focus is on not only *reproducing* the failure, but *reconstructing* it—that is, to devise a self-contained sequence of events that reproduce the failure. Unlike record/replay, however, we aim at avoiding all overhead at runtime.

2.4 Execution Synthesis

The development of RECORE was mainly inspired by recent work on *execution synthesis*. Execution synthesis [123] is a technique that automatically reproduces bugs in C/C++ programs starting from a bug report, *without doing any recording while the software is running in production*.

Execution synthesis uses the core dump to extract the program failure condition and the stack trace (but not the heap). It combines static analysis, symbolic execution [67, 15] and thread schedule synthesis to find an execution that matches the stack trace and the failure in the core dump. This combination of analyses reduces the search space of possible executions. The dynamic search of execution synthesis is based on Klee [15]: it uses context-sensitive proximity heuristic to select the executions that are more likely to reproduce the failure. Once it finds a path that reproduces the failure, it uses Klee to compute the concrete program inputs required to reproduce the failure.

The main challenge of execution synthesis is reducing the search space. This is especially problematic for deep execution paths that generate hard-to-solve constraints. In such cases, execution synthesis will take longer to reproduce a failure or timeout.

RECORE goes beyond execution synthesis in a number of ways. First, it is more general: While execution synthesis only uses the final stack trace, RECORE also relies on method arguments and targets—and can be easily extended to arbitrary mid-execution events or states. Second, by using a *search-based* approach to reconstructing the core dump, it can produce partial solutions—that is, test cases that reconstruct as much of the failure as possible, but which need not satisfy *all* constraints imposed by a constraint-based approach such as execution synthesis. On the other hand, execution synthesis is applicable to multi-threaded programs and can deduce inputs that would be hard to find through evolutionary search. In the long run, we expect test case generation (and thus failure reconstruction) to integrate evolutionary algorithms with constraint solvers to form a greater whole.

2.5 Defect Localization

Defect localization is concerned with automatically identifying defects in the source code. As it lies at the heart of automated debugging; this is a broad and diversified field.

A general problem is that, as discussed earlier, there are infinitely many ways to fix any given defect. So in the general case, automatically localizing a defect is futile. However, when applying some heuristics (e.g., that code changes should be minimal) or under certain assumptions (e.g., that the code is “almost” correct), current defect localization approaches can amazingly accurately pinpoint problems in the code. There are different approaches to reach that goal, some of which try to work in a fully automated fashion and some of which try to help the developer.

Agrawal and colleagues [3] created a tool termed χ slice that produces dices (set difference of dynamic slices) to localize defects. Renieris and Reiss [99] try to localize the fault by comparing the coverage spectra of a failing run to its “nearest” passing run (according to their distance metric). Pytlik and colleagues [96] do preliminary work with two types of simple variable invariants. Given a counterexample trace of a model checker, Groce’s Explain tool [42] creates a passing run with minimal distance and extracts a specialized dynamic slice. Dallmeier and colleagues [22] use method call sequences to identify defective classes. Qi and colleagues [97] use the path conditions of a correct and a faulty version of a program to generate an additional execution path and present differences as possible locations of the defect. The Deputo tool of Abreu and colleagues [1] combines a spectrum-based approach with model-based diagnosis.

2.5.1 Statistical Debugging

Statistical debugging techniques are a family of techniques that identify potentially faulty code by observing the characteristics of a large number of failing and passing program executions. Intuitively, entities that are exercised mostly by failing executions are more likely to be faulty than entities that are exercised mostly by passing executions. The first statistical debugging approach, Tarantula, uses statement coverage information to assign suspiciousness values to statements and rank them accordingly [64]. Liblit and colleagues extended Tarantula by defining a more general approach that focuses on predicate values instead of statements [75, 76]. Chilimbi and colleagues [17] use path profiles instead. Since then, a large number of statistical debugging approaches have been presented in the literature (e.g. [2, 44, 77, 78]). These approaches differ from one another in the type of entities they consider (e.g., statements, predicates, data values) and in the type of statistical analysis performed on the information collected for such entities.

Our BUGEX approach employs statistical debugging techniques to calculate the correlation between individual facts and the failure. Specifically, it makes use of the approach of Abreu and colleagues [2]: Every fact is treated as an independent component of the system. A diagnosis d_k and an observation obs are sets of such components. Specifically, an observation is a set of components that participated in a certain execution—either a passing or a failing one. The components in a diagnosis represent possible causes for the failure, and thus the diagnosis has a certain probability to explain a set of (both failing and passing) observations. The ranking is calculated as the sum of the conditional probability—according to Bayes’ theorem—of the diagnosis d_k given the observation obs , for all such observations:

$$P(d_k|obs) = \frac{P(obs|d_k)}{P(obs)} \times P(d_k)$$

Since $P(obs)$ is a normalizing denominator that is identical for all probabilities, it does not have to be computed directly. The A-priori probability $P(\{j\})$ that an individual component j is faulty usually depends on code complexity,

design, and other parameters. However, since determining this A-priori probability is a complex problem in itself, it is simply assumed that each component has the same A-priori probability of $p = 0.1$ of being faulty. The probability that a diagnosis d_k is correct without considering any observations, in a system with M components is therefore given by the following formula:

$$P(d_k) = p^{|d_k|} \times (1 - p)^{M - |d_k|}$$

The probability of making an observation obs given a diagnosis d_k , in most cases depends on a goodness factor $g(d_k) \in [0; 1]$. This goodness factor considers the possibility that a faulty component may also exhibit correct behavior and may therefore also have participated in the generation of correct results. The goodness factor is simply the sum of the number of failing executions divided by the sum of the number of all executions in which the component participated. The probability to make an observation obs given a diagnosis d_k is therefore calculated using the following formula:

$$P(obs|d_k) = \begin{cases} 0 & \text{if } d_k \wedge obs \Rightarrow \perp \\ 1 & \text{if } d_k \Rightarrow obs \\ g(d_k)^t & \text{else if run passed} \\ 1 - g(d_k)^t & \text{else if run failed} \end{cases}$$

In the formula, t is the number of faulty components involved in the execution according to d_k . Using the above approach, BUGEX calculates the probability that a given diagnosis is correct.

The approach of Abreu and colleagues [2] as described above computes minimal diagnoses to rank and thus is also well-suited for situations with multiple defects. However, the calculation of minimal diagnoses as minimal hitting sets is an NP-complete problem [98] and thus the use of the deterministic approach is prohibitive. Better performing approximations [1] generated diagnoses that did not work well for us. But where Abreu and colleagues were interested in locating defects, we want to explain them. So we have to question the underlying assumption that we can exclude all other but minimal diagnoses—and think about situations where multiple branches are *together* relevant for a failure (e.g., because they encode an or-condition).

A main issues with statistical debugging techniques is that they require a large number of test cases to work well. Given the right heuristics, a small number of test cases can be sufficient to narrow down the potential culprits to, say, 5–10% of the code [118], but this can still be hundreds of unrelated code lines. To pinpoint a bug to a handful of lines, one may need hundreds or thousands of additional test cases, which unfortunately are rarely available in practice. Even if an extensive test suite were to exist, the tests in it might not have the right characteristics (e.g., they might not have enough discriminating power). In most practical cases, in fact, the starting point of a debugging session is a single failing test case.

BUGEX needs only a single failing test to operate: starting from a failing run, BUGEX systematically generates both failing and passing tests whose executions are used to confirm or refute possible correlations between observed facts and the failure at hand. In addition, BUGEX can steer the test generation so as to target interesting facts and further accentuate differences.

2.5.2 Experimental Debugging

The second line of automated debugging techniques is characterized by their *experimental* approach. Rather than assuming a set of executions, these techniques *generate* additional executions whose outcome guides the systematic isolation of failure causes. The first representative of these techniques is *delta debugging*, an approach starting with only one failing and one passing run and isolating minimal failure-inducing differences in inputs [127], code changes [124], or program states [125]. The most recent work in the field minimizes object interaction [14], to a test case encompassing a mere 0.2% of the source code. Most related to our work is the concept of *predicate switching*, an approach that flips branch outcomes [129] during execution in order to identify defect-related branches.

As they operate on a potentially unlimited number of executions, experimental approaches can narrow down failure causes with high precision. However, they manipulate executions in a way that may be unsound and may thus raise infeasibility issues. Such infeasibility may in turn compromise diagnoses and make it harder to understand the results when they involve state or branch manipulation.

BUGEX is inherently experimental in nature, as it systematically generates test cases to isolate failure causes and uses the outcome of previous tests to generate new ones. BUGEX therefore also assumes the presence of an *oracle*—that is, an automated means to distinguish passing from failing runs. This can be as simple as a single assertion or even no oracle at all in case of program crashes. However, in contrast to techniques such as delta debugging or predicate switching, it is sound, and the executions it generates and uses for its diagnosis are always feasible.

2.6 Program Understanding

Before the developer can conceive a fix, they have to understand the problem. However, the concrete form of this process mainly depends on the developers previous knowledge of the system. The whole scientific field of *program understanding* is concerned with this process, but many works assume that the programmer has *no* previous knowledge of the system. For our work we assume that gaining contextual knowledge of the design and purpose of the system are preconditions, without which the understanding of the actual problem cannot take place. But even a programmer who has perfect understanding of the design of the system and its intended functionality, may still have problems understanding where the difference between intended and actual behavior arises from. However, when it comes to specialized maintenance tasks, theories on large scale program comprehension are still in their infancy [112].

Brooks [13] presumes that the general programming process consists of constructing mappings from a problem domain, possibly through several intermediate domains, into the programming domain. Thus comprehending an existing program involves reconstructing part or all of these mappings. Furthermore, he

assumes that this reconstruction process is driven by the creation, confirmation, and refinement of hypotheses. Which is exactly what BUGEX does.

The main goal of BUGEX is to help programmers understand how failures come to be. BUGEX presents different *facts* about the execution to the user. The intuition of these facts is comparable to the facts LaToza and colleagues [72] observed when monitoring developers as they tried to understand foreign code. This notion is mirrored in other contexts as “spectra”, “features” or “execution profiles”. Harrold and colleagues [47] give a very good overview and classification of the different possible types of program spectra. Santelices and colleagues [106] find that for a set of lightweight fault-localization techniques, a combination of different types of program spectra (such as we use) outperforms individual approaches.

The main problem of approaches that aim to foster program understanding is that helpfulness is not a measurable quantity, so it is not directly and objectively verifiable whether the approach meets its goal. To verify such an approach, one has to conduct human studies, which tend to be time-consuming, laborious and error-prone. Perhaps this is one of the reasons, why this goal has received so little attention in the past.

Most automated debugging techniques discussed so far focus on getting good *quantitative* results in predicting defect locations and are often evaluated using unrealistic benchmarks (see Section 2.8). However, they provide no explanation or context for why a given statement is ranked as suspicious. These techniques simply assume *perfect bug understanding*, that is, that simply examining a faulty statement in isolation is enough for a developer to detect, understand, and fix the corresponding bug. Unfortunately, perfect bug understanding rarely occurs in practice, as developers need additional information to recognize and correct bugs [94].

Only a few approaches explicitly try to help the developer understand a failure. Tarantula, for instance, visualizes test information and highlights suspicious code [64], but provides no further explanation. The cause-effect chains and cause transitions of Zeller [125] and Cleve and Zeller [19] explain a failure in terms of how variable values cause each other through a program execution, but ignore other important aspects. Zhang et al. [128] generate comments for a given failing test case by changing variable values of the test and associating these values with the execution outcome. They then generalize over these values using a rules based invariant detection engine. In contrast, BUGEX allows to correlate arbitrary runtime facts with the failure, including branches taken (implying the statements executed) or variables taking specific values. The probabilistic program dependence graphs (PPDG) from Baah and colleagues [9] can also be used to comprehend a fault, but since it bases on intraprocedural program dependence graphs, it does not work across functions. Rapid, by Hsu and colleagues [49] use a string matching algorithm to identify common segments in the traces of failing executions. Context-Aware Statistical Debugging by Jang and Su [59] combines statistical debugging, clustering, and control-flow analysis to identify relevant control flow paths that may contain bug locations.

Dialog-oriented approaches also explicitly support failure understanding. Whyline is a tool from Ko and colleagues [69] that allows developers to ask

questions about the visual output of a program based on a recorded execution. The tool by Hao and colleagues [45] suggests breakpoints to the developer for an interactive localization of the defect. BUGEX could also be used in such an interactive setting. However, our evaluation results suggest that its precision is high enough to render further interaction unnecessary.

2.7 User Studies

The common evaluation scheme [99] used to rank automated debugging techniques, bases on some assumptions that a recent user study by Parnin and Orso [94] showed several issues with.

1. The evaluation assumes that the developer will proceed linearly through long lists of unrelated suspicious locations. However, this contrasts the findings of the user study, in which successful developers *jump* through such lists, and only developers that were unable to fix the defect proceeded in a linear fashion.
2. Success is measured in the percentage of the code of the whole program, the developer would have to inspect before reaching the defect in the ranking. In contrast to that, the user study showed that, after a certain number of statements inspected, the user’s confidence in the results ceased and the users stopped using the tool. That number was *absolute* and not related to the size of the program under investigation.
3. The evaluation assumes *perfect bug understanding*, which means showing the defective line of code in isolation suffices for the developer to recognize the defect. Again, the user study showed that developers need some explanation or context to understand the defect before being able to recognize it in the code.

Also, Parnin and Orso [94] found something else during their user study: developers fix the same bug in *different locations*. However, while the authors simply assumed that some developers chose the “wrong” place to fix the bug, we think reality is much more intricate: we assume that there is no “right” location of the bug in the first place. Instead, the bug can be fixed equally well in multiple locations. The whole notion of the *location of the bug* also stems from a flaw in the evaluation scheme: in all evaluations that we are aware of (both user studies and quantitative evaluations), either a given correct program is changed to *introduce* (seed) bugs, then the “correct” fix is the one that reverts the change. Or the “correct” fix is the solution to the bug as recorded in a bug database, which we deem merely an accidental selection of the infinitely many possible fixes. And even more important, we assume that, often enough, the fix of a bug consists of more than a single line of code and can even be distributed over several locations in the code. This takes the whole notion of the “location of a bug” ad absurdum.

Instead of trying to locate the bug, we should aim to *help the developer understand the bug*—which simultaneously addresses all problems mentioned

above. However, as bug understanding is not a directly measurable quantity, this makes the evaluation problematic.

To the best of our knowledge, only a few studies with actual users of automated debugging tools were conducted. Weiser [114] performed a study on slicing with 21 programmers and programs whose sizes ranged from 25 to 150 lines of code. Francel and Rugaber [31] performed a study with 17 students, also on slicing. Kusumoto and colleagues [71] also performed a study on slicing with 6 students and programs of around 400 lines of code. Sherwood and Murphy [107] performed an experiment on a debugging tool that shows differential coverage between execution traces with six subjects and real-world medium-sized programs. Ko and Myers [69] performed an extensive empirical study evaluating their Whyline tool with 20 subjects and a program of 150 thousand lines of code.

2.8 Bug Benchmarks

In order to properly evaluate automated debugging techniques as discussed above, one needs to apply them to well documented defects [109, 87] that are already fixed, so that results can be verified. In order to allow to draw conclusions as general as possible about the performance and applicability of the approach, an evaluation should consider as many different real defects from as many different real projects as possible. The general intention is that the defects form a representative set of the different types of defects and that the projects come with a context (e.g., size, number of tests, ...) as realistic as possible. However, present bug benchmarks often use very small or specific projects with an unrealistic amount of test cases, and still lack in the amount and variety of defects—often they even contain seeded (intentionally inserted) defects.

Most of the approaches discussed in Section 2.5.1 are evaluated on the Siemens Suite [46, 50]. The Siemens suite consists of seven C/C++ applications that also have been ported to JAVA for benchmarking purposes. The Siemens suite is unrealistic due to the small size of its programs, manually crafted one-line defects, and the excessive size of its test suite, which favors statistical approaches. Most of the defects are semantic and some defect types (e.g., such as concurrency defects) are missing entirely. And since the code was ported from C, none of the applications are object-oriented.

The *PEST benchmark* [80] suffers from similar short-comings as the Siemens suite but contains only two subjects. *BugBench* [79] contains defects from seventeen open source C/C++ projects. The contained defects are mostly memory-related—only a small amount of defects are related to concurrency or semantics. The *Software-Artifact Infrastructure Repository* (SIR) [26] contains a growing number of C/C++ and JAVA programs. Most applications are relatively small and most bugs are artificial [23]. *iBugs*[24] contains 390 real defects from three open-source JAVA projects. Although this are many defects, the small number of projects may impair its generalizability.

In summary, none of the presented bug benchmarks manages to fulfill the requirements of an evaluation of automated debugging tools applicable to object-oriented JAVA code, in that they do not offer a representative collection of real defects from a multitude of different projects of varying size and purpose. This is one of the reasons, why our evaluation was conducted with only seven defects. However, as detailed in [Section 5.2.1](#), we aim to remedy these issues.

Chapter 3

From Failure to Test

Parts of the contents of this chapter have been published in Rößler et al. [104].

The basic idea of RECORE is to leverage *search-based test generation* (Section 2.1), to produce a test case that triggers *as similar a failure as possible*. As discussed in Section 1.2, reproducing the identical failure is impossible and anyway of no practical value. So here, “similarity” means similarity of the *stack trace* (the active methods) as well as similarity of the *memory dump* (local variables and method arguments) at the moment of the failure. Guided by such a dump (also called “core dump”), our RECORE test case generator searches for a series of events that precisely reconstructs the failure from primitive data. We can thus reconstruct the *entire history* of the failure-inducing state:

1. RECORE¹ is the first technique to use *search-based test generation* to reconstruct failures from saved core dumps. Being search-based, it can also produce partial solutions. Furthermore, the approach can be easily extended towards reproducing other recorded events; to have search-based test generation reconstruct failures, one only needs a fitness function and a seeder leveraging post-mortem data. Both features are in contrast to constraint-based approaches such as execution synthesis [123].
2. RECORE² goes beyond existing record/replay tools such as RECRASH [8] or BUGREDUX [60] in that the approach *requires zero runtime overhead*.³ That is, production code can run unchanged, with same time and space requirements; RECORE kicks in only at the moment of failure to produce a core dump.
3. Finally, rather than unserializing data and objects from the core dump, as RECRASH does, RECORE *reconstructs all objects from primitives*. The resulting test case thus shows a history of how the failing data comes to be.

¹“RECORE” stands for “Reconstructing core dumps”.

²“RECORE” is also one bit beyond “RECORD” (as in record/replay).

³This assumes regular binaries. For JAVA programs and other interpreted languages, we assume support of the virtual machine to produce core dumps, as discussed in Section 3.2.1.

3.1 An Example

To exemplify how RECORE works, we will use the JODATIME Brazilian Date example introduced in [Section 1.1](#). RECORE uses an adapted version of EVOSUITE [34] to search for a test that triggers as similar a failure as possible, with the criteria as defined in [Section 1.2](#). However, to create a guidance for the evolutionary algorithm that RECORE bases on, we consider additional factors when calculating how close we are to our goal (blended into the fitness function as detailed in [Section 3.2.2](#)).

First, we analyze the stack trace (as given in [Figure 1.4](#)) and the memory dump. In our evaluation ([Section 3.3](#)), we started off with existing manually created test cases, so we simply assumed that the stack trace is relevant in its entirety. However, in a real world environment, we would likely receive stack traces which contain additional layers (e.g. GUI) of code that is irrelevant for the developer. In that case, the developer would have to configure on which layer of the stack trace they would want the unit test to base on.

Then we create some initial test cases. EVOSUITE does this by randomly choosing a method from the project to call. We adapted this approach such that with decreasing probability, RECORE choses a method from the stack trace, any method of an object from the stack trace, or a method of an object on the core dump.

We will use an ongoing example to illustrate how such an evolution could look like. However, it is important to note that all decisions shown during the evolution of this example will only occur with a certain probability, as this is a random-based approach and not a deductive one.

For one of the many initial test cases created, RECORE could choose the `toInterval` method of the `org.joda.time.LocalDate` class (because it is a method on the stack trace as shown in [Figure 1.4](#)). Then RECORE tries to create an object to call the method upon. So it tries to create an object of the type `org.joda.time.LocalDate`. This class happens to have 11 constructors, 2 public static methods that indirectly create an instance, and there are 3 other methods within the project that return a `org.joda.time.LocalDate` object.

For the sake of this example, we assume RECORE happens to choose the constructor which takes a single parameter of type `long`. Now RECORE needs to come up with a value for this parameter. We tweaked EVOSUITE such that with decreasing probability, it either chooses primitive values as found on the memory dump from within an object of the class for which the value will be a parameter, a value from the memory dump or a random value. With a certain probability, RECORE could chose that parameter to be 0. To be able to call that method, RECORE also needs to come up with a parameter of the type `org.joda.time.DateTimeZone`. With a certain probability, RECORE could decide to use `null` as parameter. So RECORE just randomly created the test case as shown in [Figure 3.1](#).

RECORE executes this test case twice. In the first run it determines where in the execution the call stack of the test gets closest to the given target stack

```

1 void recore_test() {
2     LocalDate localDate0 = new LocalDate(0L);
3     Interval interval0 = localDate0.toInterval(null);
4 }

```

Figure 3.1: Initial test case as generated by RECORE

trace, and in the second run it triggers a memory dump at that very location. Analyzing the results, RECORE recognizes that, a) the test case does not throw any exception at all, b) that it does not even get close to the target location, and c) that the differences in the memory dump are substantial. Among others, it notices that the `iLocalMillis` field in the `org.joda.time.LocalDate` object differs. In the target memory dump, this field has the value `1255824000000`, whereas in the memory dump of the generated test case, that field has the value `0`.

Since the result is not yet satisfactory, RECORE now randomly decides to mutate this test case. With a certain probability, RECORE could decide to replace the `long` parameter with a different value. Because it recognized the difference in the memory dump, now it will use the value as found on the original memory dump with a much higher probability when looking for values of that type. So chances are high that RECORE decides to use the value as found.

```

1 void recore_test() {
2     LocalDate localDate0 = new LocalDate(1255824000000L);
3     Interval interval0 = localDate0.toInterval(null);
4 }

```

Figure 3.2: Test case as generated by RECORE after feedback

RECORE might decide to replace the `null` parameter as well. This parameter is of type `org.joda.time.DateTimeZone`. This is an abstract class, and there are 6 public static methods which return an object of this type. RECORE might use the `forTimeZone` method of the `org.joda.time.DateTimeZone` class. For this, it needs an object of type `java.util.TimeZone` as a parameter. The class `org.joda.time.tz.FixedDateTimeZone` provides a method, which returns an object of this type: `toTimeZone()`. So now RECORE needs to instantiate the class `org.joda.time.tz.FixedDateTimeZone`. This class has a single constructor and there exists only one other method in the project which returns an object of this type. With a certain probability, RECORE could choose the constructor. For this, RECORE needs to come up with different primitive values in the same manner as indicated above. So now the test case looks as shown in [Figure 3.3](#).

When executing this test case, RECORE recognizes an improvement both in terms of the produced stack trace as well as in terms of the produced memory dump, so it prefers this mutated version over its predecessor. Since the parameter of the `toInterval` method has changed, RECORE is now able to

```

1 void recore_test() {
2     LocalDate localDate0 = new LocalDate(1255824000000L);
3     FixedDateTimeZone fixedDateTimeZone0 =
4         new FixedDateTimeZone("zsgsdasr", "", 7, 7);
5     TimeZone timeZone0 = fixedDateTimeZone0.toTimeZone();
6     DateTimeZone dateTimeZone0 =
7         DateTimeZone.forTimeZone(timeZone0);
8     Interval interval0 =
9         localDate0.toInterval(dateTimeZone0);
10 }

```

Figure 3.3: Mutated test case as generated by RECORE

compare the internals of the two parameters as represented in the memory dumps. Among others, it recognizes a difference of the `java.lang.String` value of the `iID` field, where in the target memory dump, the field has the value `"America/Sao_Paulo"`, and in the memory dump of the generated test case, the field has the value `"zsgsdasr"`. So again, when trying to find a new `java.lang.String` value for the test case, this value has a much higher chance of getting chosen.

This generated test case so far still does neither fully reproduce the stack trace nor the memory dump nor does it even throw an exception. So another round of mutations is applied. This time, RECORE decides to change the first argument of the `org.joda.time.tz.FixedDateTimeZone` constructor and use the value supplied by the memory dump difference. The resulting test case is shown in [Figure 3.4](#). Again, note that this is not deduced but merely a random decision. However, in such a case, deduction might be a valuable future improvement.

```

1 void recore_test() {
2     LocalDate localDate0 = new LocalDate(1255824000000L);
3     FixedDateTimeZone fixedDateTimeZone0 =
4         new FixedDateTimeZone("America/Sao_Paulo",
5             "", 7, 7);
6     TimeZone timeZone0 = fixedDateTimeZone0.toTimeZone();
7     DateTimeZone dateTimeZone0 =
8         DateTimeZone.forTimeZone(timeZone0);
9     Interval interval0 =
10        localDate0.toInterval(dateTimeZone0);
11 }

```

Figure 3.4: The test case generated by RECORE reproduces exactly the stack trace in [Figure 1.4](#).

This test case fails like the original test case (see [Figure 1.3](#)), reproducing exactly the same stack trace (as shown in [Figure 1.4](#)). The test has all the necessary input:

- `localDate0` holds a specific instant in `JODATIME` format: 1,255,824,000,000 milliseconds after January 1, 1970. This corresponds to October 18, 2009.
- `fixedDateTimeZone0` holds the time zone: "America/Sao_Paulo".
- Feeding this date and time zone into `toInterval()` raises an exception.

Developers can now load this test case into the debugger and reproduce the failure at will; as it is self-contained, no additional context (such as serialized objects from the original core dump) is required. Better yet, one can feed the test case as is into an automated debugger. The `BUGEX` tool [102], for instance, would pinpoint the failure-inducing branch, showing that October 18, 2009 is the last day of daylight savings time (DST) in Brazil, a condition that is improperly handled by `JODATIME` due to an inconsistent mapping between UTC time (the internal intermediate format) and local time. Applying the official fix for the original failure also fixes the `RECORE` test case, indicating that it indeed triggered the exact defect.

3.2 Implementation Details

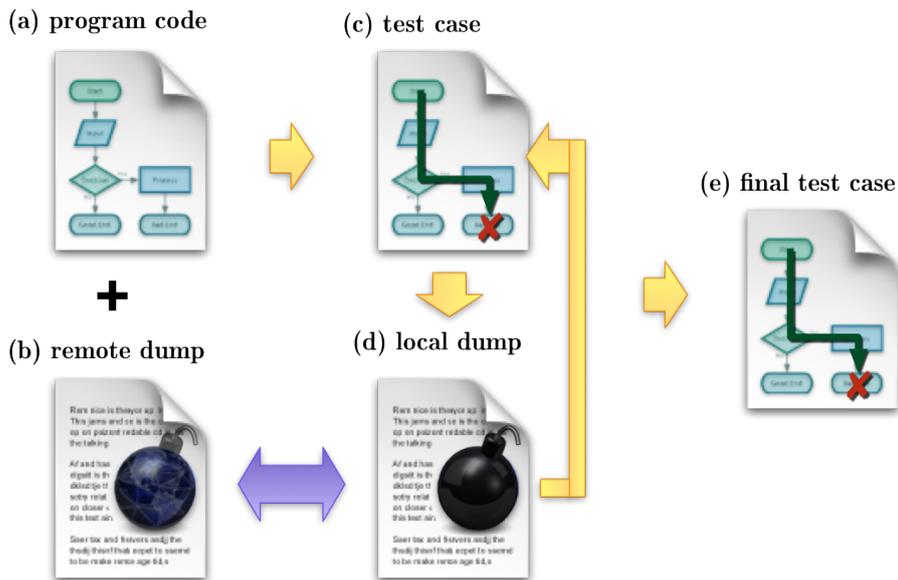


Figure 3.5: Overview how RECORE works

Figure 3.5 shows the basic steps of RECORE. It starts with an executable program (a) and a remote core dump (b), consisting of stack and heap data. RECORE then uses `EVOSUITE` as test generator, leveraging the core dump as fitness guidance and source for primitive values, to generate tests (c) whose final state (d) is then compared to the original core dump (b). The test generator systematically evolves the test case to increase similarity. When the original stack trace is reconstructed, RECORE issues the appropriate test case (e).

3.2.1 Collecting Crash Data

Operating systems such as Unix or Windows can be set up to produce a core dump upon abnormal termination of the program—a file (traditionally named “core”) which would contain the program counter, stack pointer, memory contents, and other processor and operating system information. Such a core file can be loaded into a debugger to explore the state at the moment of failure just as if the program were still running. As core dumps are only produced in case of failures, they induce zero overhead in regular runs.

RECORE is set up to work on JAVA programs. In JAVA programs, the equivalent of abnormal termination is an *unchecked exception*—that is, conditions that cannot be reasonably recovered from, which generally cause abnormal termination, and which indicate bugs to be fixed. Such an exception contains the stack trace at the moment of failure, but does not contain memory information; in particular, local variables and arguments at the time of failure are lost as the exception is raised through the stack. In analogy to the execution of regular binaries, one would normally set up the JAVA virtual machine to produce a core dump whenever an unchecked exception is raised—again only incurring overhead in case of fatal failures.⁴

Our current RECORE prototype does *not* extend the JAVA virtual machine; instead, it instruments the code such that a raised exception captures the heap, all local variables, and all method arguments at the time of failure. To do this for every method on the call stack, RECORE wraps each method in a try-catch block. This is done on Bytecode level, using JavaAgent technology [53] to instrument the client code when it is loaded. So only the binaries of the program are needed. However, to illustrate this, Figure 3.6 shows what it would look like in JAVA source code.

```

1  try {
2  // method code as found
3  } catch (RuntimeException exc) {
4      exc.pushStackFrameDump("methodName",
5                             this, new Object []{<parameters>},
6                             new Object []{<localVariables>});
7      throw exc;
8  }

```

Figure 3.6: The try-catch that is put around each method

This approach likely induces an overhead on most JAVA virtual machines. This potential overhead, however, is only due to the current implementation; if RECORE were written for C programs, or if RECORE would plug directly into the JAVA virtual machine, any program would be executed without any change to execution time or memory—until a fatal failure occurs, which is when dumping the current state sets in.

⁴Generally speaking, unchecked exceptions simply should not occur. Even if an unchecked exception would be caught and handled during execution, such that execution resumes, one would still be interested in reproducing the failure—and also in the resulting core dump.

The adapted code as shown in [Figure 3.6](#) simply adds the method’s name, the callee, the current argument values and the current values of the local variables to the exception. Since storing this information is no standard exception functionality, we provide a slightly changed `java.lang.Throwable` in the JAVA bootclasspath [56]. That class offers the additional functionality of two methods which are inherited by all subclasses (see [Figure 3.7](#)).

```

1  pushStackFrameDump(String method,
2                          Object callee,
3                          Object [] parameters,
4                          Object [] localVars);
5  Stack<Object []> getStackDump();

```

Figure 3.7: Methods added to the `java.lang.Throwable`

Finally, the main class of the code is instrumented to set our own implementation of a `java.lang.Thread.UncaughtExceptionHandler` (as shown in [Figure 3.8](#)). This handler uses standard JAVA MXBean technology [55, 54] to create a heap dump. This dump also contains the thrown exception which holds the gathered information about parameters and local objects.

```

1  static {
2      Thread.setDefaultUncaughtExceptionHandler(
3          new HeapDumpingExceptionHandler());
4  }

```

Figure 3.8: Code added to the client’s main class

In case the exception is not caught and would thus crash the program, this handler is called.

By default, RECORE is set up to record the state only in case of *unchecked exceptions*—that is, conditions that cannot be reasonably recovered from, which generally cause abnormal termination, and which indicate bugs to be fixed. With this setting, we mimic core dumps of regular binaries, and thus only incur overhead in case of fatal failures. RECORE can also be set up to record the state for non-fatal exceptions (such as invalid user input, database problems, missing files, etc.). Storing the current stack frame is a very small overhead compared to the overall effort in raising an exception, and exceptions (as the name suggests) would be expected to be rare, such that we would expect the overhead to be hardly measurable in practice. But as the promise of RECORE is zero overhead at runtime, this zero overhead is the default setting.

3.2.2 Fitness Function

The key idea of RECORE is to set up a *fitness function* that checks similarity to guide the evolutionary algorithm. Generally spoken, the better the fitness

of a particular test case, the likelier it is that it will influence the generation of future test cases. Hence, defining an appropriate fitness function is the central part of RECORE. Note that we cast this as a minimization problem and set up the genetic algorithm to minimize the fitness—so a fitness of 0 is an ideal result.

Statement Distance

The first factor that determines the fitness of a test case is its ability to exactly reproduce the given *stack trace*. For this, we need a measure of how close we are in reaching a stack trace, for which in turn we first need a measure of how close we are in reaching a specific location in the code.

For a given *test* and a location *l* in the code (in our case: class, method, and line), we assume a function $StatementDistance(test, l)$ whose value is the higher the farther away the *test* is from reaching *l*. A value of zero means zero distance (i.e. *l* was reached); a value of one means maximal distance.

$StatementDistance$ can be based directly on a fitness function as used in search-based test generation [86]. In our case, $StatementDistance$ is based on the unchanged fitness function from EVOSUITE, the platform which RECORE uses to generate tests. The EVOSUITE fitness function uses two measures as guidance:

- The **approach level** [113] determines how close the generated test is in reaching the target code (typically, yet uncovered code). It denotes the distance between the target point in the control dependence graph and the point at which execution diverged: The higher the approach level, the more control dependent branches between test and target.
- The **branch distance** [70] determines how close the branch predicate at the point of diversion is to evaluating to the desired value. The branch distance can be calculated using a simple set of rules [86]: For instance, given a branch condition `if (x < 15)`, the value `x = 20` implies a higher branch distance than `x = 16`.

These two measures are combined as follows:

$$StatementDistance(test, l) = approach\ level(test, l) + norm(branch\ distance(test, l))$$

To normalize the branch distance in the range $[0, 1]$, we use $norm(x) = x/(x + 1)$.

Stack Trace Distance

Based on *StatementDistance*, we can now define stack trace distance. We denote a stack trace $S = \langle l_1, \dots, l_n \rangle$ as a sequence of source code locations l_i , where l_1 is the outermost frame, l_n the innermost frame, and each frame l_i is invoked from the frame l_{i-1} .

Let $R = \langle r_1, \dots, r_n \rangle$ be the reference stack trace from the core dump, and $S = \langle s_1, \dots, s_m \rangle$ be the stack trace from the generated test. Let

$$lcp = \max\{j \in [1, \dots, \min(n, m)] \mid \forall i \leq j : r_i = s_i\}$$

be the length of the longest common prefix of R and S , starting from the outermost caller. We define the function *StackTraceDistance*(R, S) as follows. If $lcp = |R| = |S|$ holds, then *StackTraceDistance*(R, S) = 0 holds as well, as the stack traces are identical. Otherwise, we define it to depend on the number of stack frames yet to reach (expressed by the value of $|R| - lcp$) and the distance between the test and the first diverging stack frame r_{lcp} :

$$\begin{aligned} \textit{StackTraceDistance}(R, S) = \\ |R| - lcp - (1 - \textit{norm}(\textit{StatementDistance}(\textit{test}, r_{lcp}))) \end{aligned}$$

By using *StackTraceDistance* as guidance in the fitness function, test case generation will thus strive to maximize stack trace similarity, starting from the outermost frames.

Stack Dump Distance

The second factor that determines the fitness of a stack trace is its ability to reconstruct variable values as found in the core dump. In principle, we could guide RECORE to reconstruct *all* variables from the heap and from the stack. However, such an attempt is likely to be misleading. First, we would have to reconstruct thousands of individual values, which is unlikely to be feasible. Second, few of these values actually matter for reproducing and fixing the failure. We thus decided to focus on those values which we also require to reconstruct a given stack trace, namely *those values found in the stack*.

We assume a function *ObjectDistance*(x, y) $\in [0, 1]$ which compares two objects x and y and again returns a normalized value between zero and one. We obtain this value by relying on the EVOSUITE object distance, which is based on the comparison of primitive field values:

- If x and y contain **numbers**, the normalized absolute difference is added to the total difference.
- If x and y contain **strings**, the normalized Levenshtein distance [74] is added to the total difference.
- If x and y contain **complex objects**, the field values of these objects are compared recursively, adding the result to the total difference.

- If either field value is **null**, 1 is added to the total difference.

The total difference is then divided by the number of fields.

Based on *ObjectDistance*, we can now define the *stack dump distance* across all objects as found on the stack. Let $obj(s)$ denote the set of object identifiers in local scope at a location s (i.e., the currently active object, all method parameters and all local variables). Let $s(o)$ denote the object identified by o . We then define the *stack dump distance* as the sum over all object distances as found on the common stack frames:

$$StackDumpDistance(R, S) = \sum_{i=0}^{lcp} \sum_{o \in obj(r_i)} ObjectDistance(r_i(o), s_i(o))$$

Again, the larger the distance, the more and the higher the object differences.

Total Fitness

During the execution of a generated test, we obtain several stack traces, each with its own stack dump distance. To measure the fitness of a test, we go for the *minimum of all traces obtained*. Let S_1, \dots, S_n denote the stack traces obtained during the execution of a test *test*. We can then define test *trace* and *dump distances* as follows:

$$TestStackTraceDistance(R, test) = \min\{StackTraceDistance(R, S) \cdot S \in \{S_1, \dots, S_n\}\}$$

as well as

$$TestStackDumpDistance(R, test) = \min\{StackDumpDistance(R, S) \cdot S \in \{S_1, \dots, S_n\}\}$$

Again, the higher the similarity, the lower the fitness value.

We give the highest priority to reconstruct the stack trace, whereas reconstructing the local variables is a lesser goal. Hence, the stack trace distance determines the overall fitness, whereas the stack dump distance is in $[0, 1]$. This is also helpful in guiding test generation while no progress is made reconstructing the call stack.

Finally, we impose a penalty for runs where the target exception is not thrown and thus install some lock-in effect: once the exception was thrown by a test, other factors (i.e. stack dump distance) cannot guide test case generation away from the exception again.

$$ExceptionPenalty(test) = \begin{cases} 0 & \text{if same exception is thrown} \\ 1 & \text{otherwise} \end{cases}$$

The sum of these three gives the fitness function whose value RECORE strives to minimize:

$$\begin{aligned} \text{Fitness}(R, test) = & \text{TestStackTraceDistance}(R, test) \\ & + \text{TestStackDumpDistance}(R, test) \\ & + \text{ExceptionPenalty}(test) \end{aligned}$$

RECORE extends EVOSUITE with the fitness function above, thus guiding test case generation towards reconstructing the given core dump. The genetic algorithm stops once the best known fitness remains unchanged for 1,000 generations.

3.2.3 Seeding Primitive Values

A core dump is not only helpful for guiding the search. It also contains all the primitive values found on the heap at the point of failure. These values can be used to *seed* the search—that is, provide useful starting points rather than values taken at random.

In evolutionary search, *seeding* refers to any technique that exploits previous related knowledge to help solve the problem at hand. Although in general seeding should not be a requirement to solve the problem at hand, it can boost the search such that a solution is found with limited search budget, where this would be impossible without. Seeding is also an important component of EVOSUITE [33]; for example, EVOSUITE re-uses primitive values and strings found in the bytecode.

RECORE sets up EVOSUITE seeding to take advantage of the core dump. Note that the following probability values were chosen arbitrarily. We believe that changing these parameters may affect the time it takes to search for these results, but not necessarily the result quality.

1. **Seeding values.** To seed values, RECORE *only* uses values from the heap dump (as the dump also contains all constants from the source code).
 - With a probability of $p = \frac{1}{3}$, RECORE uses the *feedback* it receives from the distance function (Section 3.2.2): If a primitive P in the reference stack dump is different from the current best test case, then P is directly fed back to be used.
 - With $p = \frac{2}{3}$, RECORE uses *a value from the heap dump*: If the value is to be used in a method or constructor that occurs in the stack trace, then it reuses the parameters for that method or constructor from the stack dump. If the method or constructor stems from a class that is found somewhere on the stack dump, RECORE reuses the primitive values it finds as the field variables of that class. Otherwise, it uses an arbitrary value from the heap dump.

2. **Creating method calls.** When inserting or replacing method calls or constructors in a test case, EVOSUITE chooses randomly from the set of known calls.

- With $p = \frac{3}{7}$, RECORE makes EVOSUITE use one of the calls on the stack trace. This way, we give preference to the methods that need to be on the stack trace in the end.
- With $p = \frac{2}{7}$, RECORE uses any method or constructor of the classes on the stack trace.
- With $p = \frac{1}{7}$, RECORE uses the *feedback* it receives from the distance function (Section 3.2.2): If an object in the reference stack dump is different from the current best test case (i.e. different class or non-null), then a method or constructor from the class of that object is used.
- Eventually, with $p = \frac{1}{7}$, an arbitrary method from any class (of the target project) that can be found on the heap is used.

Both the above measures dramatically speed up failure reconstruction and cause RECORE to make the same use of core dumps as a human debugger doing a post-mortem analysis. Rather than just seeding values, the core dump would of course also allow us to simply bypass most of test case generation by *taking objects from the core dump* and feeding these into the failing methods. As discussed in Section 2.3, this can result in very precise reconstruction of the failure per se; however, the *history* of how the objects got into their final state is lost. As we are interested in reconstructing the failure history from the beginning, RECORE only uses primitive values from the core dump.

In RECORE, we go for a middle ground instead: As it comes to synthesizing objects, we always resort to invoking the official constructors. *Primitive* arguments (numbers, strings, booleans, enumerations), however, are selected from values found in the core dump. This way, RECORE also addresses the *provisioning problem* of test case generation: Normally, such primitive values would have to be randomly constructed. By using values from the actual execution, we can produce much more realistic test cases—in particular, test cases that are similar to the original failure.

3.2.4 Limitations

As a test generator, RECORE suffers from the general limitations of such tools. To start with, there is no general constructive way of reaching a specific program state—this is an instance of the halting problem. Test generation tools such as RECORE will thus fail to reconstruct a failure if the search space is ill-formed or the conditions are very complex. In a situation in which only few specific inputs lead to the desired state (such as passwords in cryptographic checks, for instance), RECORE is unlikely to construct helpful inputs.

These general concerns are offset by the fact that test case generation often achieves high coverage in practical settings. In our specific setting of reconstructing core dumps, we even *know* that the specific state we search is reachable. In

Table 3.1: RECORE evaluation subjects

ID	Section	Subject	Lines of code
JOD1	3.3.1	Brazilian Date bug	62,326
VM1	3.3.2	Vending Machine bug	68
MAT1	3.3.3	Sparse Iterator bug	53,496
JOD2	3.3.4	Western Hemisphere bug	62,326
JOD3	3.3.5	Parse French Date bug	53,845
COD1	3.3.6	Base64 Decoder bug	8,147
COD2	3.3.7	Base64 Lookup bug	6,154

principle, we can therefore reconstruct any state simply by having RECORE search long enough—but this is only a theoretical option. Additionally, the core dump may contain precisely those specific inputs we are looking for—but they may just as well have been overwritten by later computations before the failure occurs.

For all these reasons, it is *unrealistic to assume that RECORE will always be successful*. It may be useful and effective in practice in a number of situations, though. Whether this is the case on real-life programs with real-life failures will be explored in the next section.

3.3 Case Studies

To evaluate the potential of RECORE, we applied it to a set of seven bugs, summarized in [Table 3.1](#). Our selection of programs and defects was driven by the following requirements:

1. The underlying test case generation technique (EVOSUITE, in our case) must be able to handle the program and defect.
2. The failure must not depend on artifacts other than the program and the corresponding test case.
3. The defect must be well explained, documented and must come with a patch/-fix, such that we can validate the result of the approach.

Note that these are fairly lightweight requirements. The first one only requires that (a) the test case that reveals the defect can be encoded in EVOSUITE’s internal format and (b) EVOSUITE can generate inputs for the program. The second and third ones are fairly standard requirements.

Our experiments are promising: RECORE is able to reconstruct the exact failure in five cases, and partially in another case. The resulting test cases are self-contained, short, and comprehensible. At the same time, the RECORE approach incurs no overhead in production code. Because it operates fully automated, this means that the resulting test cases come at literally not cost.

Each of these bugs is produced by a single test case which we set up such that a core dump would be produced upon failure (Section 3.2.1). We then fed RECORE with the original program as well as the core dump. What we wanted to know were the answers to three questions:

- **Can RECORE produce test cases that produce the exact same stack trace?** Note that this is one of RECORE’s success criteria, so this question may also be posed as: “Does RECORE produce a result?”
- **Does the generated test case pass after applying the original fix?** This question asks: “Is the test case useful in debugging the original failure?” It should actually run the other way round: If we fix the error based on the produced test case, would we also fix the original failure? For the original failure, however, we have the single official fix, so we use this one as ground truth.
- **Are the test cases readable and easy to understand?** The answer to this question is best left to the reader. For this purpose, we provide a detailed analysis of each failure and test case; this is also the reason why we prefer seven in-depth case studies to statistical summaries over bug collections.

Table 3.2 provides the running time of RECORE for the seven subjects.⁵ In the remaining sections, we present the seven reconstructed failures.

3.3.1 JodaTime Brazilian Date Bug

Our first bug is the Brazilian Date Bug discussed in Section 1.1, namely bug report 2487417 for JODATIME [62]. It takes RECORE 35 minutes to reconstruct this failure. The stack trace matches exactly, and the official fix also makes the test case pass. This is a poster example for the capabilities of RECORE—reconstructing the exact time and time zone in which the error occurs.

Interestingly, the only major difference between the generated test case and the human-written test case is that the generated test case (Figure 1.3) generates the date from the number of seconds (see Figure 3.9), where the human-written test case generates the date using a calendar date (Figure 3.10).

Table 3.2: Summary of results

ID	Section	Duration	Reproduces stack?	Passes after fix?
JOD1	3.3.1	35 m ± 17 m	yes	yes
VM1	3.3.2	9 m ± 2 m	yes	yes
MAT1	3.3.3	17 m ± 18 m	yes	yes
JOD2	3.3.4	22 m ± 14 m	no	no
JOD3	3.3.5	18 m ± 12 m	yes	no
COD1	3.3.6	149 m ± 108 m	yes	yes
COD2	3.3.7	6 m ± 1 m	yes	yes

```
1  localDate1 = new LocalDate(1255824000000L);
```

Figure 3.9: Date in number of seconds

```
1  localDate1 = new LocalDate(2009, 10, 18);
```

Figure 3.10: Date in calendar form

Why can't RECORE use the "nicer" calendar date? The number of seconds is contained in the `LocalDate` object in the heap, whereas the calendar date constituents would have to be searched and reconstructed first. Finding such inputs is within the capabilities of evolutionary algorithms, but RECORE will go with the simpler solution first.

3.3.2 Vending Machine Bug

Vending Machine is a small artificial example used for earlier studies on automated debugging [14, 102]. A vending operation that can cause the credit to fall below zero is erroneously enabled, raising an exception when invoked. RECORE reproduces a test that precisely captures the failure (Figure 3.11).

```
1  VendingMachine vendingMachine0 = new VendingMachine();
2  String string0 = "SILVERDOLLAR";
3  Coin coin0 = Coin.create(string0);
4  vendingMachine0.insert(coin0);
5  vendingMachine0.vend();
6  vendingMachine0.vend();
```

Figure 3.11: Test generated by RECORE for the Vending Machine bug

Again, the stack trace matches exactly, and applying the official fix makes the test case pass.

3.3.3 Commons Math Sparse Iterator Bug

Apache Commons Math is a library of lightweight, self-contained mathematics and statistics components. Defect number 367 [83] is a `NullPointerException` raised by a *sparse iterator*, which should iterate over the non-zero values in a vector. The test case generated by RECORE reconstructs the failure (Figure 3.12).

⁵All times were measured on a quad-core 2.67 GHz Intel x86 CPU with 8 GB RAM; RECORE and EVOsuite are single-threaded.

```

1 ArrayRealVector arrayRealVector0 = new ArrayRealVector();
2 double double0 = -1832.3093176108437;
3 ArrayRealVector arrayRealVector1 =
4   (ArrayRealVector) arrayRealVector0.append(double0);
5 ComposableFunction composableFunction0 =
6   ComposableFunction.SIGNUM;
7 RealVector realVector0 =
8   arrayRealVector1.map(composableFunction0);

```

Figure 3.12: Test generated by RECORE for the Sparse Iterator Bug

Interestingly, this test case produces a stack trace that *extends* the original—that is, it has more callers on the outside. Rather than calling the failing iterator functions directly, RECORE reproduces the failure indirectly as it finds that the method `map()` by itself invokes the iterator and triggers the failure. Again, the generated test case passes as the official fix is applied.

On a related note: After applying the fix, the given test fails with a different exception (i.e. the correct exception), whereas the generated test passes.

3.3.4 JodaTime Western Hemisphere Bug

In JODATIME bug report 2889499 [63], selecting a time zone at the border of the Western hemisphere and calling `toDateTimeZone()` can cause an arithmetic exception after a long chain of subsequent time/date calculations. Due to the complex dependencies, RECORE fails to reach the statement at which the error was raised, and thus cannot reconstruct the issue.

With this test case, we have met the limitations discussed in [Section 3.2.4](#); a better test case generator may be able to address this issue. Note, though, that this failure comes at virtually no cost: The attempt to reproduce the error is fully automatic, and the production code has no runtime overhead.

3.3.5 JodaTime Parse French Date Bug

In the JODATIME bug report 1788282, parsing a legal french date fails with an `IllegalArgumentException` [61]. The test generated by RECORE reproduces the exact stack trace ([Figure 3.13](#)).

With this, however, RECORE has not recreated the original failure. The input `"11.sept..2007"` is correctly reconstructed, and the stack trace is the same. However, the failure is different:

- In the original run, the date format is `"dd.MMM.yyyy"`. Here, `"MMM"` stands for the possibly abbreviated month, which in French can include a dot (i.e., the `"sept."` in `"11.sept..2007"`).

```

1 DateTimeFormatter dateTimeFormatter0 =
2     ISODateTimeFormat.time();
3 Locale locale0 = Locale.FRENCH;
4 DateTimeFormatter dateTimeFormatter1 =
5     dateTimeFormatter0.withLocale(locale0);
6 String string0 = "11.sept..2007";
7 Date date0 =
8     dateTimeFormatter1.parseDateTime(string0);

```

Figure 3.13: Test generated by RECORE for the Parse French Date Bug

- The test case generated by RECORE uses an ISO date format, which is "yyyy-MM-dd". The provided input "11.sept..2007" obviously does not match the ISO format; hence parsing fails.

The difference in failure manifests itself in a different state; the original run fails to parse the month, whereas the generated run fails at the first dot. This also results in different exception messages, which instructs the programmer to proceed with caution.

Why can't RECORE reproduce the original failure? The `DateTimeFormatter` class takes a string in its constructor. To recreate the object in the core dump, RECORE must pass the original string "dd.MMM.yyyy". However, the fact that this string was used in the constructor is lost in history, and reverse-executing the parser to reconstruct it is beyond RECORE's capabilities (and actually, beyond the capabilities of any of today's test generators).

With this test case, we thus again have met the limitations discussed in [Section 3.2.4](#). It should be noted, though, that half of the input (namely, the french date) is correctly reconstructed, which should speed up manual debugging considerably.

3.3.6 Commons Codec Base64 Decoder Bug

The *Apache Commons Codec library* provides implementations of common encoders and decoders such as Base64, Hex, Phonetic and URLs. Issue 98 [21] reports that "certain (malformed?) input to `Base64InputStream` causes a `NullPointerException` in `Base64.decode()`." The description of the problem as found in the bug report is given in [Figure 3.14](#).

The test case generated by RECORE reconstructs this failure as given in [Figure 3.15](#). The input in this test case (`US_ASCII`) differs from the test case which triggered the original failure; in particular, a character encoding ("`US-ASCII`") would normally not be used as a string to encode. However, this generated input (8 characters) is much smaller than the reported input (1190 characters). The test case produces the exact same stack trace; and again, applying the original fix also makes this test case pass.

Certain (malformed?) input to Base64InputStream causes a NullPointerException in Base64.decode(). The exception occurs when Base64.decode() is entered with the following conditions:

- *[The field] buffer is null.*
- *[The field] modulus is 3 from a previous entry.*
- *[The parameter] inAvail is -1 because Base64InputStream.read() reaches EOF on line 150.*

Under these conditions, Base64.decode() reaches line 581 with buffer still null and throws a NullPointerException.

Figure 3.14: Description of the Base64 Decoder bug as found in the bug report

```

1 String string1 = CharEncoding.US_ASCII;
2 byte[] byteArray0 = Base64.decodeBase64(string1);
3 ByteArrayInputStream byteArrayInputStream0 =
4   new ByteArrayInputStream(byteArray0);
5 Base64InputStream base64InputStream0 =
6   new Base64InputStream(byteArrayInputStream0);
7 byte[] byteArray1 = Base64TestData.streamToBytes(
8   base64InputStream0, byteArray0);

```

Figure 3.15: Test case generated by RECORE for the Base64 Decoder bug

3.3.7 Commons Codec Base64 Lookup Bug

The issue number 22 [20] shows that the Apache Commons Codec library throws an `ArrayIndexOutOfBoundsException`. To reconstruct the failure, RECORE produces an array with one negative element as input, which is also the case in the original failure (Figure 3.16).

```

1 byte byte0 = -125;
2 byte[] byteArray0 = new byte[3];
3 byteArray0[0] = byte0;
4 byte byte1 = 64;
5 byteArray0[2] = byte1;
6 boolean boolean0 = Base64.isArrayByteBase64(byteArray0);

```

Figure 3.16: Test case generated by RECORE for the Base64 Lookup bug

Again, the stack trace is identical and the original fix also fixes the generated test case.

3.3.8 Threats to Validity

As any empirical study, our evaluation is subject to threats to validity.

Threats to *construct validity* have to do with how we measured the effectiveness of our technique. We assume a failure to be reconstructed if the stack trace is the same, and if the original fix makes the generated test case pass. For a full evaluation of effectiveness, we would have developers design fixes based on the RECORE results, and see how effective these fixes are in addressing the original issues. This is part of our future work (Section 5.2); for now, assessing the usefulness and readability of the generated tests is left to the reader.

Threats to *internal validity* might come from how the study was performed. We carefully tested RECORE to reduce the likelihood of defects. To counter the issue of randomized algorithms being affected by chance, we ran each experiment multiple times; all reported results are representative. The running time is the average over all runs.

Threats to *external validity* concern the generalization to software and faults other than the ones we studied, which is a common issue in empirical analysis. Our sample size is small; only seven different programs and bugs were used in the study. The reason for this is that it is time consuming to find and reproduce real bugs by manually analyzing bug reports (see also Section 2.8). This produces a bias towards well documented and easy to reproduce issues; given the general limitations of test case generation (Section 3.2.4), there will be a number of failures which RECORE will not be able to reproduce.

RECORE and its underlying EVOSUITE test case generator have several parameters such as weights, timeouts, and thresholds which all may influence the result. Wherever possible, we picked default values as used in other studies. Given the apparent effectiveness of search-based test generation in reconstructing failures, we believe that changing these parameters may affect the time it takes to search for these results, but not necessarily the result quality.

3.3.9 Summary

Our results are summarized in Table 3.2. In five out of seven issues, RECORE is able to recreate the precise failure: The generated tests reproduce the stack trace, and the original fix also fixes the test. Even when failure reconstruction is not complete (Section 3.3.5), the generated test still partially reconstructs the input.

All generated test cases reconstruct the failure from scratch, using primitive values only, and are all short enough to not only ease reproduction, but also debugging in itself. All this is achieved with zero overhead at runtime, and in an all-automatic run after the failure.

Chapter 4

From Test to Explanation

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.

— Brian Kernighan

A major effort of the debugging activity goes into *understanding* the failure. If we want to help the developer in that task, we should try to explain the failure to him. But what makes a good explanation? What does it mean to understand a failure anyway?

4.1 The First Approach

Parts of the contents of this section have been published in Rößler et al [103].

Explaining a failure fully in an automated fashion is currently far from possible. Current approaches to this overall goal try to give the developer some helpful information in pointing them to different aspects of the failure (Section 2.6). One such aspect is the equivalence classes of the input, for which the failure occurs.

The world we live in, and which we know and are accustomed to, is analogue and continuous. Within the computer, this is different. Everything is digital, using discrete, discontinuous, quantized values. And each of these values could, in principle, be treated differently. To overcome this problem and help developers with their intuitive assumption of continuous ranges, our goal was to capture discontinuities or *equivalence classes* of the input: value boundaries, for which the behavior of the application changes.

Therefore we first tried to come up with constraints on the input which are basically equivalence classes on the input or “boundaries” within which the failure occurs. These help the developer understand under which *circumstances*

the failure occurs: “The program fails whenever the date is a daylight cut day and the cutover time is midnight.” This eventually helps characterizing, understanding, and classifying the problem, and in turn is important to assess the severity of it and helps in identifying related or duplicate problems.

Essentially, we explore a technique to *identify failure circumstances automatically*. Starting with a concrete failing run, we first compute the *path condition* leading to the failure—a conjunct of constraints on the inputs under which the control flow reaches the location where the specific failure occurs with the right state. This conjunction of constraints provides the conditions for the failure to occur, but can become arbitrarily complex. In order to *simplify and generalize* the collected path conditions and identify the general *failure conditions*, we use a combination of systematic experimentation, constraint solving, and constraint abstraction.

4.1.1 Intuition Behind the Approach

In a recent study at Microsoft [68], knowing “in what situations a failure occurred” was the third most frequent unsatisfied information need of developers, and by far the most time-consuming of all. Why is this so difficult?

As a simple real-life example, consider the JODATIME Brazil example, as given in the Introduction (Section 1.1): It takes as input a specific date and timezone and converts the date into an 24 hours interval in the given timezone.

At this point, we have everything in place to start our investigation. We have a failing application and a test input that allows us to reproduce the failure at will, but we do not know under which circumstances the program fails. We do not know whether the failure would occur for other date and timezone combinations and, if so, for which ones. It is also possible, although unlikely, that the failure may occur for this very specific combination only, in which case we might focus on more prevalent issues first. Therefore, even before we start the actual debugging, *we need to understand the failure circumstances*.

For relatively simple programs, our approach produces a short and crisp description of the failure circumstances. For the given failure, for instance, the computed failure condition are shown in Figure 4.1.

```
1 isDaylightCutDay() and (cutoverTime == 00:00:00)
```

Figure 4.1: Failure conditions for the JODATIME Brazilian example

This means that the failure occurs whenever (1) the date is a daylight cut day in the given timezone, and (2) the cutover time is midnight.

4.1.2 Implementation Details

Figure 4.2 provides an overview of the approach: Our technique takes two inputs—a program p and an input i that makes the program fail—and produces

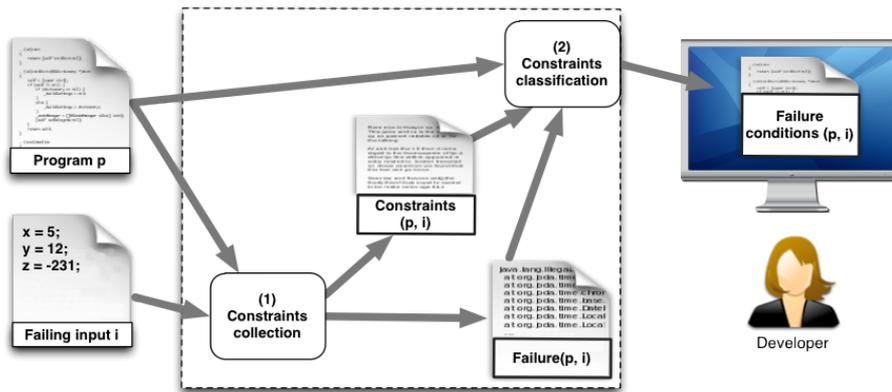


Figure 4.2: Intuitive view of the approach

one output—a set of *failure conditions* that can be examined by the developer. To produce such conditions, the technique performs two steps: (1) constraints collection and (2) constraints classification.

Phase 1: Constraints collection

Phase 1 of our technique first runs the given test to detect the specific failure under investigation and collect the path conditions (the set of constraints on the input parameters) corresponding to the failure. In the pseudocode shown in [Figure 4.3](#), which depicts the main steps of our technique, Phase 1 is summarized by lines 1–3 of the algorithm, where `executeConSym` is the function that executes the program concretely and symbolically at the same time. The constraints can be fairly complex even for a relatively simple and short fragment of code.

Phase 2: Constraints classification

After collecting the path conditions (PC) for the failing execution of interest, in Phase 2 our technique iterates over those conditions to classify them as irrelevant, enabling, or failure conditions. As shown in the pseudocode in [Figure 4.3](#), for each constraint c , the technique (a) creates an alternative set of path conditions (PC'), where c is replaced with its negation (line 5) and (b) tries to get a solution for PC' (i.e., a set of input values that satisfy PC') using a constraint solver (line 6).

If PC' is unsatisfiable, or the constraint solver timeouts trying to find a solution for it, the algorithm delays the classification of c (not shown in the pseudocode for simplicity) until some other constraints are dropped. It would have also been possible to have the constraint solver return the minimal unsatisfiable core and negate all related constraints at once. Conversely, if PC' is satisfiable, our technique reruns the program using the newly computed input values, which can result in one of three outcomes:

Parameters: program p , failing input i
Result: failure conditions C_{fail} , enabling conditions $C_{enabling}$

```

1:  $C_{fail} := true; C_{enabling} := true;$ 
2:  $o := p(i);$ 
3:  $PC := executeConSym(p, i);$ 
4: for all  $c \in PC$  do
5:    $PC' := PC - \{c\} + \{\neg c\};$ 
6:    $i' := solve(PC');$ 
7:    $o' := p(i');$ 
8:   if  $o = o'$  then
9:      $PC := PC - \{c\};$ 
10:  else if  $orac(i') \neq o'$  then
11:     $C_{enabling} := C_{enabling} \wedge \{c\};$ 
12:  else
13:     $C_{fail} := C_{fail} \wedge \{c\};$ 
14:  end if
15:   $summarizeConstraints(C_{enabling}, C_{fail});$ 
16: end for

```

Figure 4.3: Simplified pseudocode that depicts the approach

The new inputs still cause the original failure (line 8), which means that c did not affect the outcome of the execution. In this case, the technique classifies c as irrelevant and drops it.

The new inputs cause a failure different from the original one (line 10). In this case, we assume that the constraint represents a precondition that the inputs must satisfy to be valid and reach the point of the original failure. Thus, our technique classifies c as an enabling condition.

The new inputs trigger neither the original nor a different failure (line 12). In this case, our technique classifies c as a failure condition, that is, a conditions that is necessary to trigger the failure of interest.

At the end of Phase 2, our technique has identified a subset of the original PC divided into two sets of constraints: enabling and failure conditions. Finally, to summarize the constraints, our technique uses information collected during dynamic symbolic execution to (a) identify and factor out recurring expressions within constraints and (b) identify methods that return a boolean value and can be used to summarize subsets of constraints they generate (line 15). For our example, the technique would identify the constraints created while executing `isDaylightCutDay()` as constraints generated by a boolean method and would suitably replace them with the name of the method. The result of such substitution would be the short and crisp description of the failure.

As this description would be generated in a fully automated way, it could be provided to the developers at no additional (human) cost to help them understand and eliminate the failure.

```

1 public void doSomething(int x){
2     if (x < 0) {
3         throw new IllegalArgumentException(...);
4     }
5     if (x <= 5) {
6         System.out.println("x is " + x);
7     }
8     if (x < 7) {
9         throw new IllegalStateException(...);
10    }
11 }

```

Figure 4.4: Simple example to further illustrate the approach

4.1.3 Simplifying Constraints

To illustrate constraint simplification in more depth, consider the code in [Figure 4.4](#). Assume we have a test case that calls method `doSomething()` with a value 4 for parameter x . The initial step of our algorithm of [Figure 4.3](#) would execute the test and observe an `IllegalStateException` being thrown. The corresponding PC consists of the following constraints: $(x \geq 0) \wedge (x \leq 5) \wedge (x < 7)$.

The technique would then iterate over the constraints to classify them. Assume that the algorithm arbitrarily chooses $(x \geq 0)$ to be classified first. Negating it yields the following PC': $(x < 0) \wedge (x < 5) \wedge (x < 7)$. A constraint solver fed with these constraints would return a value of x that satisfies the constraints, such as $x = -1$. The technique would then execute the test again with parameter x being -1 , which would result in an `IllegalArgumentException` being thrown. Because this failure is different from the initial one, our algorithm would classify the constraint that was negated as an enabling condition and keep it in the PC. In fact, this constraint encodes the condition for an input to be valid and, although it does not trigger the failure, it is necessary to reproduce it.

Assume now that the next constraint the algorithm chooses to negate is $(x \leq 5)$, which results in the PC' being $(x \geq 0) \wedge (x > 5) \wedge (x < 7)$. Given these constraints, the constraint solver would return $x = 6$. Again, the technique would rerun the program, this time setting parameter x to 6. The outcome of this execution would be an `IllegalStateException`, which is the original failure we are analyzing. The technique would therefore conclude that the constraint is irrelevant for reproducing the failure and would drop it, leaving only $(x \geq 0) \wedge (x < 7)$ in the set of constraints. Again, it is clear from the code that this is the right course of action, as the condition guards a statement that writes to the console; for the failure to occur, it is irrelevant, whether that statement is executed or not.

The next and last constraint the algorithm chooses to negate would be $(x < 7)$, which results in PC' being $(x \geq 0) \wedge (x \geq 7)$. In this case, any value for x greater or equal to 7 would satisfy the constraints, so let us just assume that the constraint solver produces the solution $x = 7$. When the technique reruns

the program with x being 7, the result is a normal execution of the program that does not produce any failure. The algorithm would therefore classify the constraint as a failure condition, a constraint that is necessary for the failure of interest to be triggered.

At the end, the technique would present the developer with two constraints: constraint ($x \geq 0$), marked as an enabling condition, and constraint ($x < 7$), marked as a failure condition. The developer could then use this information to understand and correct the fault in the code.

4.1.4 Limitations

We have developed a prototype tool that implements our technique. The prototype is built on top of Java PathFinder (JPF) [48], a symbolic execution engine for Java programs, and leverages JPF's extensions for dynamic symbolic execution and the CVC3 constraint solver [10].

However, we encountered several practical challenges when using our tool on real-world examples (such as the one given above):

Hard-to-solve Constraints

Some generated constraints go beyond the solving capabilities of existing constraint solvers (e.g., constraints generated by caching and hashing functions that make use of bit-wise operators). This could possibly be addressed by two (possibly complementary) directions: the use of solvers that operate on formulas defined over the theory of bit-vectors and arrays, such as STP [36]; and the replacement of constraints that go beyond the theories supported by the solver with their concrete values from a previous execution.

Disjunctions

One problem we encountered for the initial formulation of our approach is related to the presence of predicates containing **OR** operators in the code. Since Java short-circuits **OR** operators (i.e., if the first condition is true, the other conditions are not evaluated), our technique may only observe a subset of a disjunctive predicate at runtime. This is an issue because, as discussed in Section 4.1.3, our technique negates constraints and drops them if the failure still occurs for inputs that satisfy the negated constraints. Consider, for instance, the code snippet shown in Figure 4.5, and assume that the original value of x is smaller than that of y .

When executing the code, the second part of the predicate would not be evaluated, and our technique would only add to the PC constraint ($x < y$). Then, when it negates that constraint and solves the modified PC, the technique might by chance obtain a value for x that is greater than 10 from the constraint solver (e.g., due to additional conditions on y). In such a case, although the constraint is violated, the predicate it belongs to (i.e., $(x < y) \vee (x > 10)$) would still evaluate to true. The technique would therefore erroneously classify the constraint as irrelevant and drop it.

```
1 public void do(int x, int y) {  
2     if (x < y || x > 10) { fail(); }  
3 }
```

Figure 4.5: Example of a predicate that contains an OR condition

Limitations of Symbolic Execution

Because our approach relies on symbolic execution, and more specifically on dynamic symbolic execution, it also suffers from some of the limitations of this type of techniques. In particular, there may be cases where the inputs that cause the failure involve complex interactions with the environment, such as access to the network, the file system, and external databases. These types of interactions are notoriously problematic for symbolic execution techniques and limit the general applicability of our approach.

Diverging Globally

One severe problem that to this date we were not able to circumvent is the following: when negating individual constraints that are close to the failure, the corresponding execution would still be “local enough”; that is execute relevant parts of the code such that the resulting path conditions and the differences to the path condition of the failing test would be meaningful. But once the negated constraints are far away from the failure (e.g., in terms of the execution stack), the corresponding executions would go astray and execute parts of the code that are totally unrelated to the problem at hand. As a consequence we were unable to come up with additional failing executions of the same failure. To make things worse, additional generated passing executions were not very helpful either, because since they would execute parts of the code that are unrelated to the failure, the resulting path conditions would be so different from the path conditions of the failure, that the difference would not be meaningful.

Although the test case for the JODATIME Brazilian example (Figure 1.3) is fairly small and compact, it executes 367 methods and a total of 5,711 individual statements. (Note that these figures consider only code that belongs to JODATIME and ignore the execution of code in the JAVA system libraries.) When we apply the above approach to the test case, it generates 89 initial constraints, some of which contain around 2,000 subexpressions. Clearly, most of these constraints are in no way local (by any means of the term). And this would probably be similar for all but the simplest examples.

While we possibly would have been able to circumvent the other challenges, this one is systematic to our approach. It means that ultimately we would have to encode the complete program in the form of constraints (which already for programs of medium size is prohibitive). In order to focus the generated additional executions on “interesting” parts of the program (i.e., target the test case generation at executing the last branch before the failure), we decided

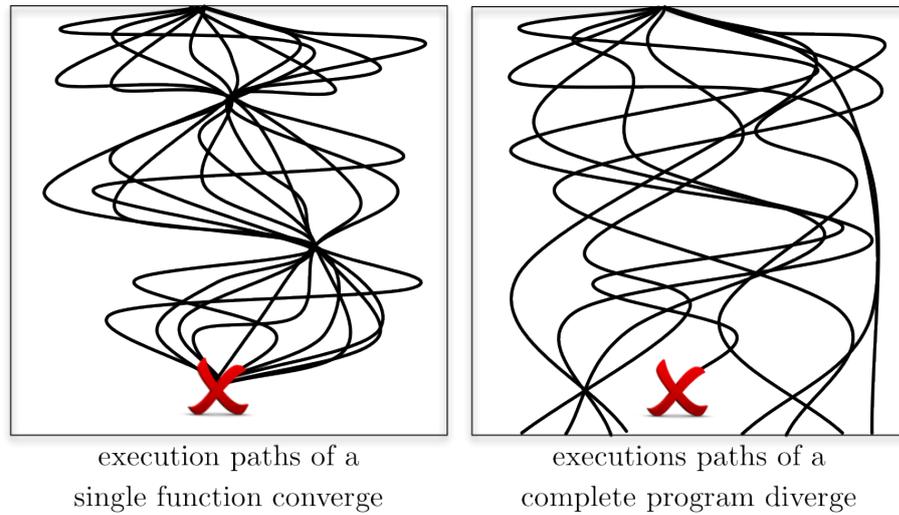


Figure 4.6: Intuitive view of how the generated execution paths relate

to employ a different execution generation strategy. We settled for an genetic algorithm (see [Section 2.2](#)). Since this is a random-based approach instead of a systematic one, we also needed to change the approach from logical deduction to a strategy that works well on a less structured and less complete set of executions. For that we settled on statistical debugging (see [Section 2.5.1](#)). Which in summary means, that we needed to change the approach completely and start all over from scratch. The approach that implements those changes is BUGEX ([Section 4.2](#)).

4.2 The BugEx approach

Parts of the contents of this section have been published in Rößler [100] and Rößler et al [102].

In this chapter, we present a novel approach to automated debugging that combines the mutual benefits of experimental and statistical approaches, yet avoids their respective shortcomings. As detailed in Section 2.5.1, one major issue of statistical debugging approaches is that they require a large body of existing test cases. And yet, most of these test cases are targeted to cover the complete project, i.e., execute as much of the code as possible *at least once*. In contrast to that, statistical debugging works best if the code in question (i.e., susceptible code) is executed as much as possible with as diverse as possible contexts. Another major issue of statistical debugging approaches is that the goal is to localize a defect. Consequently, approaches from that field fail to provide additional information to the developer to allow them to make an informed decision about whether the localization of the defect is correct or not and how the localized defect would best be fixed.

As detailed in Section 2.5.2 many approaches such as predicate switching and delta debugging on execution states (i.e. cause-effect chains) do alter executions and thereby reach states that would be impossible to reach by ordinary executions of the program. These approaches than usually rely on the runtime system to detect such states or rely on the developer to detect the error in the reasoning of the approach.

BUGEX addresses all of these issues. First, starting from a *single failing test case*, it gathers a variety of runtime *facts*. To examine the correlation of those facts to the failure, it produces additional executions that differ in as few facts as possible, concentrating on the failure at hand and on the “interesting” parts of the code—thus overcoming the shortcomings of the first approach (see Section 4.1). This way it also eliminates the need for any additional, manually crafted tests. Second, the facts that correlate with the failure tend to be related to one another and stress different aspects of the failure, such as properties of execution states or branches taken. In this way, they provide a context for each other and, ultimately, an explanation of how the failure came to be: “The failure occurs whenever the daylight savings time starts at midnight local time.” And third, since BUGEX does not tinker with the code or the executions themselves, by design it cannot create artifacts (e.g., impossible executions) that can occur using most experimental debugging techniques (such as predicate switching or delta debugging, see Section 2.5.2).

Moreover, our BUGEX approach can be seen as a *general debugging framework*, that works with any aspect of an execution that one can reason about. By systematically generating tests that confirm or refute the correlation of individual facts to the failure, it performs general systematic experimentation and thus has the *scientific approach* (observe, form hypothesis, experiment, refine) at its heart, following it in a fully automated fashion. BUGEX converges quickly and steadily and results are unambiguous and consistent with one another among different types of aspects. In fact, our results (see Section 4.2.4) show that BUGEX can pinpoint central facts with *outstanding precision and quality*. It

can be set up (e.g. on a continuous integration or build server) to start its work automatically, as soon as a failing test case is detected within the project. This way it can deliver the information at the moment a developer is informed about the failure.

In summary, our BUGEX approach

1. provides a *generic automated debugging scheme* that has the scientific approach at its heart and can be parameterized with arbitrary runtime facts (it is currently implemented with branches and state predicates, but it could also be implemented with data-flow relations, thread schedules, and more);
2. requires a *single failing run*, generating test cases as needed—unlike statistical debugging, which requires an existing test suite;
3. ensures that every run is *real* and *reproducible*, unlike state- or code-changing techniques such as predicate switching or delta debugging;
4. by *driving* the test case generation so that it highlights the differences between failing and passing runs, it pinpoints important facts with *outstanding precision and quality*, unlike statistical debugging (which relies on the capabilities of the given test suite);
5. provides a *general* explanation of the failure that is valid for *all* encountered executions and may therefore even improve over the findings of manual debugging.

In our evaluation, a prototype of BUGEX precisely pinpointed important failure explaining facts for six out of seven real-life bugs.

4.2.1 Explanatory Facts

To fully understand a failure, one has to take into account various aspects, such as input, program structure and runtime behavior both in terms of control flow and data flow. Information about these aspects is represented by *facts* of different types that complement and confirm each other and are helpful in diverse scenarios. Examples of such facts are:

Constraints on the input: The input-related path conditions, as they are encountered during execution, form constraints on that input. The proposed approach tries to extract 1-minimal [127] failure conditions that explain the failure in terms of the input (as shown by previous work [103]). For the JODATIME Brazilian example, these failure conditions are given in Figure 4.7.

```
1 isDaylightCutDay() and (cutoverTime == 00:00:00)
```

Figure 4.7: Conditions necessary to trigger the Brazilian example

That is, the failure occurs whenever the chosen date is a daylight cut day in the corresponding time zone and the cutover time is midnight.

Variable values: The relations and abstractions of the values of variables may also point to the underlying problem. For the initial example, the value of the expression given in [Figure 4.8](#) has always the same relation to the chosen date: It is the date of the next daylight savings time transition, whenever that transition is at 00:00:00. That is in line with the constraints on the input as mentioned above.

```
1 dateTZ.nextTransition(date.getLocalMillis())
```

Figure 4.8: Date of the next daylight savings time transition

State predicates: State predicates (or Program invariants [29]) that hold for failing runs, but never for passing runs or vice versa often reveal some interesting properties. For the initial example, invariants of passing runs that are violated by failing runs are given in [Figure 4.9](#).

```
1 millisLocal == millisUTC + getOffset(millisUTC)
2 millisUTC == millisLocal - getOffsetFromLocal(millisLocal)
```

Figure 4.9: Invariants violated by the failing test of the Brazilian example

This means that the local time is always the universal time plus offset, where the universal time is always the local time minus local offset. For the faulty runs, this mapping is inconsistent, since the created local time maps to some universal time, but this universal time in turn maps back to a different local time.

Definition-usage pairs: The definition and later usage of a variable in the code gives a good understanding of how the data flows through an execution of the program and what data dependencies exist. If the problematic definition and later usage of a value are far away from each other, which is especially common in object oriented programs, that connection can be important to show. For the initial example, such definition-usage pairs could be used to show the origin of the invariant violation as given above (see [Figure 4.10](#)).

```
1 getOffsetFromLocal(millisLocal) !=
2     getOffset(millisLocal -
3         getOffsetFromLocal(millisLocal))
```

Figure 4.10: Definition-usage pair for the Brazilian example

This is the guarding condition of an internal consistency check which discovers that the start of the day (midnight) does not exist in the chosen time zone and raises an exception.

Executed branches: The correlation between executed branches (branch coverage) and defects is an often chosen indicator to single out problematic code. Yet for the given example, this shows a problem that arises when only considering existing executions: the code has several places where daylight savings time

transitions are treated differently, and due to the nature of the problem, all those branches are highly correlated to the failure. Only when generating additional executions that challenge the correlation of individual branches, it becomes clear which of these branches are really relevant for the failure and which are just coincidentally correlated. The branch in the `org.joda.time.DateTimeZone` class, in lines 863–864 is highly correlated to the failure (Figure 4.11).

```

1 // if the offsets differ,
2 // we must be near a DST boundary
3 if (offsetLocal != offsetAdjusted) {
4     // ...
5 }

```

Figure 4.11: Branch correlated to the Brazilian example

And as the comment above it explains, this branch is indeed relevant not only to any daylight savings time transition, but to the situation handled incorrectly by JODATIME.

Number of loop-iterations: When it comes to loops, path conditions may not be enough information; the developer might also want to know how often a loop was executed. For example, in JODATIME, there is a piece of code where two dates from different APIs are aligned (Figure 4.12).

```

1 while (date.getDate() == dom) {
2     date.setTime(date.getTime() - 1000);
3 }

```

Figure 4.12: Interesting loop of the Brazilian example

In that situation, the developer wants to know whether the number of loop iterations is correlated to the failure. For instance, it would be interesting if the loop is always executed 3.600 times (representing one hour).

As detailed later, we implemented our approach for state predicates and branches. But since it is a universal debugging framework, we could have implemented it for any of the facts just outlined. Given this test of the JODATIME Brazilian example as shown in Figure 1.3, after a few minutes BUGEX reports a single branch in the JODATIME code—a branch that is taken *only by failing runs* and never by passing runs (see above).

4.2.2 The Approach in Detail

The basic steps of BUGEX are illustrated by Figure 4.13. BUGEX starts with a single failure (a) and generates additional runs (b) that are similar to the failing one and either fail or pass (c). Any difference between these runs in terms of the facts observed during such runs (d) (e.g., the fact that a specific branch was taken) would be linked to the pass/fail outcome. BUGEX statistically *ranks*

these differences, producing a ranked list of facts (e) that are strongly correlated with the failure. It then systematically generates more runs (b) that can either further confirm or refute the relevance of a fact. Eventually, BUGEX produces a ranked list of minimal fact differences, where the top-ranked facts reveal failure causes. This can be interpreted as the scientific method of observe, form hypothesis, experiment, refine.

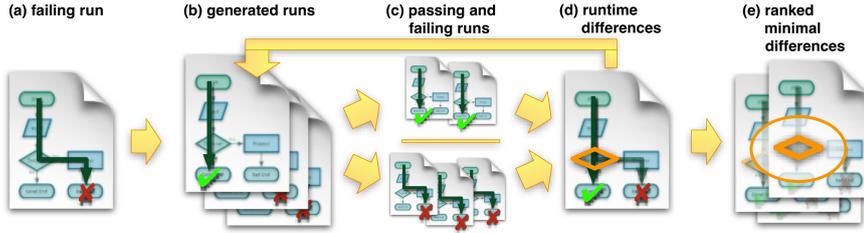


Figure 4.13: BUGEX in a nutshell

We now discuss the BUGEX approach in detail. In the following sections, we discuss the individual steps, as shown in Figure 4.13; in addition, the pseudocode in Figure 4.14 presents a schematic view of our approach.

Parameters: program p , failing test t_{fail}

Result: relevant facts $F_{correlating}$

```

1:  $b_{last} := getFailurePredicate(t_{fail});$ 
2:  $T_{fail} := \{t_{fail}\};$ 
3:  $T_{pass} := \emptyset;$ 
4:  $F := getFacts(t_{fail});$ 
5:  $F_{explored} := \emptyset;$ 
6:  $F_{correlating} := correlateToFailure(F, T_{fail}, T_{pass});$ 
7:  $F_{unexplored} := F_{correlating} \setminus F_{explored};$ 
8: while  $F_{unexplored} \neq \emptyset$  do
9:    $f := getRandom(F_{correlating});$ 
10:   $t'_{fail} := generateSimilarFailingRun(f, t_{fail}, b_{last});$ 
11:   $T_{fail} := T_{fail} \cup \{t'_{fail}\};$ 
12:   $t_{pass} := generateSimilarPassingRun(f, t_{fail}, b_{last});$ 
13:   $T_{pass} := T_{pass} \cup \{t_{pass}\};$ 
14:   $F := F \cup getFacts(t'_{fail}) \cup getFacts(t_{pass});$ 
15:   $F_{explored} := F_{explored} \cup \{f\};$ 
16:   $F_{correlating} := correlateToFailure(F, T_{fail}, T_{pass});$ 
17:   $F_{unexplored} := F_{correlating} \setminus F_{explored};$ 
18: end while

```

Figure 4.14: Simplified pseudocode that depicts the BUGEX approach

The Failing Run

We start with Step (a) in Figure 4.13, namely, the input to BUGEX. As detailed in Figure 4.14, BUGEX requires a program p and a failing test t_{fail} . (The BUGEX implementation runs on JAVA programs and requires a JUNIT test case.) Like

other experimental approaches, such as delta debugging, BUGEX requires an *oracle*—a predicate that distinguishes passing from failing runs. This oracle typically is a failing assertion, either in the test, in the code, or in the run time system. All run time exceptions (null pointers, array indexes, abnormal termination, etc.) can be debugged with BUGEX instantly, without any effort by the programmer.

```
1 object.doSomething();
```

Figure 4.15: Example code that can throw a `NullPointerException`

By default, BUGEX uses *the last branch*¹ before the failure occurs (b_{last}) as oracle (Line 1 in Figure 4.14). If that last branch is implicit it must be made explicit. As an example, consider the code given in Figure 4.15. If that code throws a `NullPointerException`, we transform this code and introduce an explicit check for `null` before (see Figure 4.16).

```
1 if (object == null) {
2     throw new NullPointerException();
3 }
4 object.doSomething();
```

Figure 4.16: Explicit `null` check in the code

This can be done automatically. All generated test cases must evaluate this last branch b_{last} ; those that do not are ignored for correlation and discarded. This last branch b_{last} encodes the oracle on the lowest abstraction level; and in most cases does not help the developer understand how the failure occurs or why that specific failure condition is met. If you consider the ubiquitous `NullPointerException`, for instance, the failure condition does not tell the developer *why* the variable was `null` at that specific point in the execution. Nevertheless, the fact that a variable *was* `null` serves well to characterize the failure itself. As this last branch correlates with failure and success by construction, it is excluded from consideration during the calculation of the correlation.

Test Case Generation

In principle, our approach can use any test generation technique that can be geared towards generating tests for different types of facts (and thus, in theory, could select the most appropriate ones for the program and failure at hand). For our current implementation, we use EVOSUITE [32], which implements an evolutionary search approach enhanced with dynamic symbolic execution (see Section 2.1). In order to create test cases, this tool is capable to generate series of method calls. However, because we start of with an existing test case, it

¹Technically, one should differentiate between a branch and the branch predicate that determines whether that branch is executed. For simplicity, and unless otherwise stated, in this work we use the term branch with both meanings.

suffices to change the values that appear in that test case. To speed up the search and avoid undesired side effects, we restrained EVOSUITE to only change values but to not change method calls. Search-based testing is well-suited for our approach, as different optimization targets can conveniently be encoded in fitness functions.

To extend EVOSUITE to support a new type of facts, one needs to (1) add a means to collect such facts for existing and newly generated test cases (e.g., via bytecode transformation) and (2) specify a fitness function that can guide test generation based on the facts of interest. In addition, the representation used by EVOSUITE needs to be capable of controlling these facts; for example, EVOSUITE is currently restricted to single-threaded programs, so it would have to be extended to be able to specify thread schedules as facts. In the current version of BUGEX, we extended EVOSUITE to support two kinds of facts: *branches* and *state predicates*.

Intuitively, the aim of test case generation is to produce tests that (1) are as similar to the failing test case as possible (see Section 1.2), but (2) differ with respect to specific individual facts. Ideally, we would find a test that evaluates the target fact differently, while at the same time evaluate all other facts the same. In fact, in that case we could logically deduce, whether the target fact is a cause. The reason for this has philosophical roots that via Lewis go back to Hume and are well explained by Groce [42]: “a cause is something that *makes a difference*: if the cause c had not been, the effect e would not have been.” This is the notion of *counterfactual causality* [126].

When changing an execution, we need to consider that such changes might introduce a new cause c' that leads to the same effect e , thus giving rise to the impression c was no cause for e . To minimize such effects, we try to have the generated executions as similar as possible. In fact, if we could generate both a failing and a passing run that would differ in only a single branch, this branch would deductively be the cause and the approach could stop immediately. However executing a branch differently in reality usually has cascading effects, such that many subsequent branches are also evaluated differently. For this reason it is unlikely that such two runs even exist, let alone that we find them using a random search approach. It is also for this reason, that deterministic straight forward approaches like delta debugging work with artificial executions and are seldom able to come up with feasible real-life examples.

Instead, for each fact BUGEX tries to generate both a passing and a failing test case that differ with respect to that fact, but possibly with respect to others as well. Therefore, in contrast to the intuitive ideal approach described above, we do not try to logically deduce the failure cause, but rather create a probability correlation for which we aim to generate helpful inputs (i.e., executions). This means that the approach is applicable even if test generation does not succeed in all cases, and for practical reasons we therefore apply a time limit to the test generation.

Test Case Requirements

During operation, BUGEX iteratively chooses a fact (Line 9 in Figure 4.14) and attempts to derive first a failing test (Line 10) and then a passing one (Line 12)

that both satisfy the following three requirements:

1. **The tests must reach the failure oracle b_{last} .** If a test does not reach the failure oracle, the resulting facts are irrelevant.²
2. **The tests must evaluate the chosen fact f differently than the original test case.** This ensures that it is possible to assess the correlation of the fact to the overall test outcome.
3. **If several test cases fulfill the above criteria, the one most similar to the original test case is selected.** As stated above, the optimal result would be a failing and passing test cases that only differ in f , which rarely occurs in practice.

The fitness function for a fact type guides the search towards satisfying these three conditions. How this guidance is implemented generally depends on the chosen type of fact, and will be illustrated in Section 4.2.3. Given such a fitness function, EVOSUITE takes care of deriving test cases that satisfy the above requirements (see [32] for details). During test generation, BUGEX *guides* and *optimizes* EVOSUITE’s search: Since EVOSUITE implements a genetic algorithm, the more different tests there are in the population to mutate and crossover, the easier it is for the genetic algorithm to come up with a test case that is good for a certain fact.

BUGEX ignores branches after b_{last} : For all test cases that we examine, we trace the execution to extract all executed branches. However there are two problems to that: First, the traces of the passing executions are usually much longer, because for obvious reasons they do not end after b_{last} . Second, the failing runs usually also do not end after b_{last} , as there is exception handling, message formatting and other code that is executed after the exception was raised. And if we later calculate a correlation between the execution of a branch and the failure, of course these branches are highest correlated, because they literally are executed always and only if the failure occurs. So in order to address this problem, we cut these executions off at b_{last} (lines 11 and 14 in Figure 4.14 respectively).

Search Optimizations

To improve the search process, we apply several optimizations:

When choosing the next fact to consider for test generation, BUGEX *ranks* the facts (see Section 4.2.2) and chooses one of the facts that is highest correlated to the failure (Line 9 in Figure 4.14). This realizes the feedback loop between test case assessment and test case generation (from (d) to (b) in Figure 4.13). This also realizes the iterative nature of the scientific method: first we observe the original failure. Then we calculate a high correlation of a fact (hypothesis). And

² In that case the execution went astray, as we already experienced in the first approach (see Section 4.1).

then we try to come up with additional executions (experiment) to strengthen, invalidate or refine the hypothesis.

In addition, this way we can achieve the quick response times shown later in [Section 4.2.4](#) because we ensure to spend the effort where it is most needed: by assessing (and attempting to refute) the high correlation of a fact to the failure. In practice, our results show that even just a few failing and passing tests are sufficient for dramatically reducing the number of relevant facts.

BUGEX uses *seeding* [33], which in evolutionary testing refers to techniques that exploit previous related knowledge to help solve the test generation problem at hand. The initial population of the search in EVOSUITE is seeded with the original failing test, as well as relevant tests collected during previous runs. In particular, a pair of test suites (T_{fail} and T_{pass}) with tests that received the highest fitness for different facts is continuously maintained (Lines 10 and 12 in [Figure 4.14](#)) and used to seed the initial population each time the test generation process is restarted. To make better use of the pool of available tests, we use a timeout when searching for tests for a fact, and repeat the search several times for every highly correlated fact f . The statistical correlation, in turn, is calculated only on the maintained body of the tests with the highest fitness value.

Although EVOSUITE is capable to evolve tests into completely new ones, to speed up the search we restrict it to only change the input values used within the tests, without changing the sequence of calls to the system.

Ranking Facts

The retrieval of runtime facts as indicated by the *getFacts* function in [Figure 4.14](#) (Lines 4 and 14) needs to be implemented for each type of facts. Our implementation for branches and state predicates works by instrumenting the bytecode of the system under test. When ranking branches, during execution of a test case, for each branch it is recorded whether the branching condition was executed and, if so, whether the branch was taken or not (or both, in case of multiple executions). When creating state predicates for a method, the values of all variables, fields and parameters are recorded on entering that method, and state predicates are created as binary relations between these values.

The generated failing and passing executions (T_{fail} and T_{pass}) are used to create a correlation between all executed facts and the failure (Lines 6 and 16). Since BUGEX's goal is to explain the fault, rather than localizing it, this correlation also considers all facts in the passing executions. By doing so, the explanation of the failure might also be a *missing* fact (e.g., the execution of a branch in which a variable is set to a value other than `null` to avoid a `NullPointerException`).

BUGEX computes the correlation of the relevant facts to the failure using an implementation of the approach by Abreu and colleagues [2] (see also [Section 2.5.1](#)). In their approach, every branch is treated as an independent component of the system. A diagnosis d_k , as well as an observation obs , are sets

of such components. Specifically, an observation is a set of components that participated in a certain execution. The components in a diagnosis d_k represent possible causes for the failure, and thus the diagnosis has a certain probability to explain a set of observations. Intuitively, this approach produces a ranking using the sum of the conditional probability, according to Bayes' theorem, of the diagnosis d_k , given the observation obs , for all such observations. Finally, our approach normalizes to one these probabilities over all diagnoses and obtains a list of facts, ranked by their *normalized probability* to be responsible for the failure.

Using this approach, BUGEX calculates the probability that a given diagnosis is correct. Currently, BUGEX generates diagnoses by considering a single component (i.e., fact) at a time. This showed to produce good results and has the additional benefit that it is computationally cheap. A minor drawback of this approach is that it only correlates *single facts* with the failure. In situations where multiple facts are, *together*, relevant for a failure, their high correlation is split among them, which would result in lower individual rankings. However, as can be seen in the Commons Math example (Section 4.2.6), since the correlation is so unambiguous, even such a split cannot introduce significant noise.

In our experiments, we usually found the ranking produced by BUGEX to be unambiguous, with top-ranked facts being ranked higher than the rest by orders of magnitudes. In order to decrease noise, BUGEX only returns the top-ranked facts and cuts the result off where the difference in ranking between two successive facts exceeds an order of magnitude.

As an example, consider the list of ranked branches for the JODATIME Brazilian Date bug. The topmost branch at line 864, shown in Figure 4.11, has a normalized probability of 0.99924, whereas the second highest ranked branch (at line 867) has a probability of 0.000043. As the difference exceeds an order of magnitude, BUGEX only reports the first branch in this case (see Figure 4.11).

4.2.3 Implementation

As detailed above, BUGEX is an approach that is applicable to any kind of fact that one deems of interest. To demonstrate and evaluate the approach, we implemented it for two types of facts: *executed branches* and *state predicates*.

Isolating Branches

In general, the similarity between two tests in our context is defined by the number of facts on which they differ. In the case of program branches, we consider the executed branches up to the failure predicate b_{last} ; execution after the failure predicate is ignored. This is necessary, since passing executions, because they did not fail, will by definition differ arbitrarily from failing executions after the point of their failure. However, these differences are not interesting for our purposes. Because b_{last} may be executed multiple times in such execution, we need to determine which execution of b_{last} matches the execution in the corresponding failing run. To do this, we use dynamic time warping [90], a technique

originally defined for the alignment of audio tracks, that matches two sequences by “warping” them non-linearly in the time dimension. Dynamic time warping lets us (1) align the traces of the passing and the original failing run and (2) cut the passing execution at the point that matches the last occurrence of the branch corresponding to b_{last} in the failing execution.

To guide the search towards evaluating the branches such that the similarity is increased, we use the *branch distance* [86] measurement, which is commonly applied in search-based testing. The branch distance estimates how close a predicate of a branch was to being evaluated in a certain way. For example, if the branch predicate ($\mathbf{arg} < 0$) is evaluated with $\mathbf{arg} = 3$, the branch distance to *false* is 0, and the branch distance to *true* is -4 . (Note that, as EVOSUITE works on Java bytecode, branching conditions are always atomic, such that there are no conjunctions or disjunctions. Programs written in other languages can easily be transformed in the same way.) Overall, the fitness of a test consists of three parts:

1. Branch distance to evaluating b_{last} as either failing or passing,
2. Branch distance to evaluating the predicate of the target branch b differently than in the original failing run,
3. Sum of the branch distances (normalized in the range $[0, 1]$) of all remaining branches to evaluating as in the original failing run.

Each of these three components is normalized in the range $[0, 1]$, and the overall fitness is the weighted sum of the three. The experimentally determined weights (4:2:1) reflect the priority of the three requirements.

Isolating State Predicates

This type of fact considers *state predicates*: predicates that can be expressed on the program state and the inputs at method entry. In the rest of the work, for simplicity and when not ambiguous, we use the term predicates to refer to state predicates.

The retrieval of the predicates is based on work of Fraser and colleagues [35]: On entry of the currently investigated method, the values of all variables, fields, parameters and constants that are in scope are recorded; complex objects are recursively resolved to accessible inspector methods and primitive values. Then, all collected values are compared with each other, using all sensible comparison operators that apply to the type of values (e.g., for integer values, this includes $==$, $<$, $>$, $<=$, $>=$). Some insensible comparisons, such as comparing constant with constant are excluded.

The resulting set of predicates represents the concrete predicates that hold for the values at method entry for one particular execution of the method. Since this equals several cross products of all considered values, it can clearly result in a large amount of predicates to consider. For the JODATIME Brazilian bug

introduced in [Section 1.1](#), for instance, on individual methods this results in well more than ten thousand predicates to be correlated with the failure.

Intuitively, the set of methods to be considered consists of all methods in the dynamic slice for b_{last} in the failing execution. In other words, all methods that contain at least one statement that was relevant for the execution path to reach b_{last} should be considered. Considering only methods in the dynamic slice often tremendously reduces the amount of relevant statements. But because such statements are scattered throughout the code, the number of relevant *methods* often is only slightly reduced. Therefore, for some programs this approach can lead to an insensibly large number of predicates, given that the developer would have to investigate the predicates for each method separately and, for several technical reasons, we have to run our approach on each method individually.

For the JODATIME Brazilian bug introduced in [Chapter 1](#), for instance, this would mean we would have to investigate the predicates generated for 82 methods. With a timeout of 30 minutes per method, in the worst case this would take about 2460 minutes or 41 hours. And if for each method we only retrieve 3 predicates, this would still yield an insurmountable 246 predicates to manually review. We thus needed to further reduce the number of methods for the approach to be practical. To do this, we focus only on methods that a) contain highly ranked branches, because these methods already showed to be relevant for the failure, or b) are on the stack trace, because, as was illustrated in [Chapter 3](#), the call stack mostly also is relevant for the failure. Changing this focus can be subject to future work.

The distance function for an individual fact can again be calculated using a metric similar to the one used for the branch distance: It is the distance of the specification variables to a target value, called *value distance*. For example, consider the predicate $(x > \text{CONST_5})$, with x being any variable, field, or parameter, and CONST_5 being a constant from the source code whose value is 5. If x has value 7 in a test case, the distance to negating this predicate is 2. As in the case of branches, the overall fitness is calculated as the weighted sum of the normalized distances.

Before presenting the result to the user, the set of highly correlated predicates is checked for implications. If one predicate implies another, the implied predicate is removed from the resulting set of predicates. This means that we keep only the most general predicates in the set.

4.2.4 Evaluation

In our experience, we found branches to be better failure indicators than the more generic state predicates. Hence, we assume that programmers would *first* consult failure-related branches, and only *later* examine failure-related state predicates—either as an *alternative* to branches, if the branches are not helpful, or in *addition* to branches, to gain more information about the conditions under which the failure occurs. This is the approach we followed in our qualitative evaluation

Real-Life Case Study

ID	Subject	Lines of code	Given tests
JOD1	Brazilian Date bug	62,326	3,497
VM1	Vending Machine bug	68	1
MAT1	Sparse Iterator bug	53,496	1,580
COD1	Base64 Decoder bug	8,147	1,149
JOD2	Western Hemisphere bug	62,326	3,497
COD2	Base64 Lookup bug	6,154	185
JOD3	Parse French Date bug	53,845	3,392

Table 4.1: BUGEX evaluation subjects³

To assess how well BUGEX works in practice, we implemented it for the two types of facts that we discussed earlier—branches and state predicates—and conducted a case study with the same subjects as the evaluation of RECORE (Section 3.3).

We deliberately chose to evaluate a small number of defects only, as to be able to report and discuss the BUGEX results for every single defect. Table 4.1 provides an overview on the program and defect characteristics.

Evaluation Setup

When conducting our study, we focused on the following research questions:

RQ1. *Is the number of relevant facts identified by BUGEX small enough for a developer to examine?*

In order to answer RQ1, we ran BUGEX on each of the seven defects and checked whether there would be a small set of branches (ideally one branch) and a small set of state predicates that would set themselves apart from the crowd.

RQ2. *Do the facts identified by BUGEX help the developer understand the failure?*

Obviously, bug understanding is not a directly measurable quantity. To answer this question, we present BUGEX’s results for each of the seven defects considered and discuss how they relate to the actual defects based on our own experiences in fixing it. We then compare our understanding to the “official” fix from the change history.

ID	Branches	Duration	Predicates	Duration
JOD1	1	2,380 s	25	13,55 s
VM1	1	19 s	1	56 s
MAT1	8	216 s	9	10,267 s
COD1	1	214 s	23	1,339 s
JOD2	7	8,422 s	9	30,937 s
COD2	1	38 s	2	737 s
JOD3	15	1,577 s	n/a	n/a

Table 4.2: Number of facts reported by BUGEX

4.2.5 Quantitative Results

Numbers of Facts Reported

Let us start with RQ1. The results produced by BUGEX can be found in [Table 4.2](#): The columns “Branches” and “Predicates” show the number of branches and state predicates reported by BUGEX as being related to the failure.

As discussed earlier, we expect developers to focus on branches first, whose number is usually in the single digits. In every case, this is a low absolute number of branches, and thus the answer to RQ1 is clearly “yes”.

For all seven defects examined, BUGEX reports a small number of branches as the failure cause; in four defects, in particular, it reports a single branch.

In terms of *predicates*, we also obtain a strong reduction, as BUGEX isolates less than 1% of all predicates to be relevant for the failure. Yet, with up to 25 state predicates, the absolute number is not as low as for branches. As we detail in [Section 4.2.6](#), however, the results fall in one of two cases: in one case, we can ignore the predicates altogether because the single branch reported already pinpoints the failure; in the other case, the top-most ranked predicates suffice to fully characterize the failure conditions. Because the predicates are generated per method, we could return only those that repeat for several methods or the predicates of methods where there were less than a certain threshold.

Time Required

One may assume that a search-based approach, where a large number of test cases have to be generated and executed, would take a considerable amount of time. Indeed, the BUGEX runtime reported in [Table 4.2](#) ranges from a handful of seconds to multiple hours.⁴

³Note that different numbers of lines of code for the same project are due to different revisions of that project.

⁴All experiments were conducted on a non-dedicated MacBook Pro, 2.53 GHz Intel Core 2 Duo, 4 GB 1067 MHz DDR3 RAM. The approach is multi-threaded.

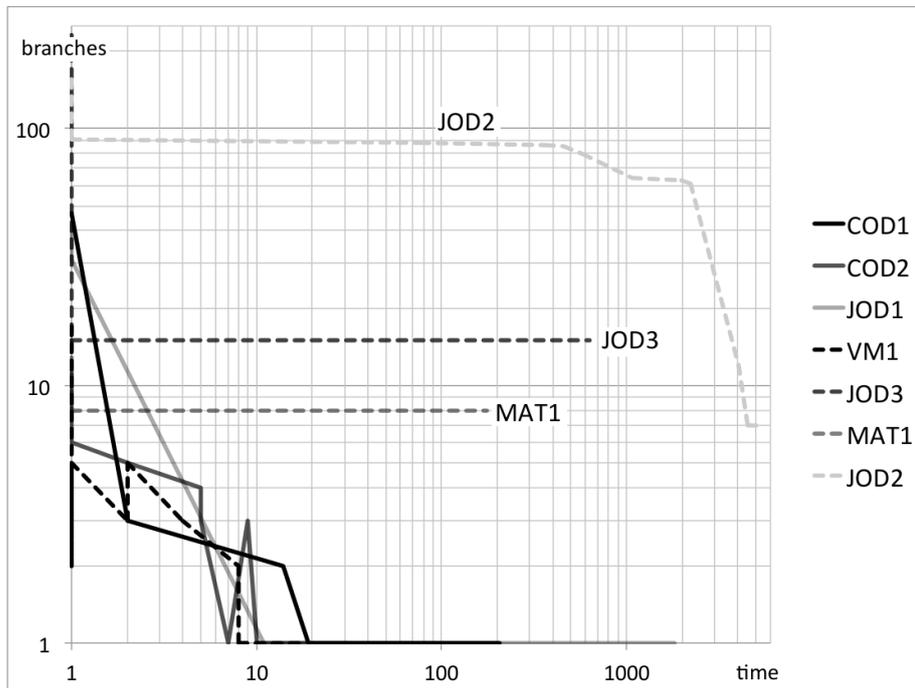


Figure 4.17: Number of branches remaining over time. The X axis reports time in seconds, the Y axis the number of branches. Note the logarithmic scale on both axes.

In practice, however, developers do not have to wait this long for their results. In Figure 4.17, we have traced the number of failure-related branches over the runtime of BUGEX. It is clear to see that for six out of seven subjects, after only 20 seconds, the number of branches (and actually, also the set of branches itself) stays stable for the remainder of the runtime. Developers may thus go for the BUGEX results after a few seconds and get all the relevant facts. Most importantly, BUGEX is not an interactive tool, so developers could simply run it overnight on the failures they need to investigate and look at the results in the morning, or configure a build server to start BUGEX as soon as a test fails.

*In six out of seven defects examined,
BUGEX was able to isolate the relevant branches after 20 seconds.*

Statistical Debugging with Supplied Test Suites

To put these numbers into context, we applied the statistical debugging approach on which BUGEX relies (see 4.2.2) on the branches executed using the test cases that were supplied with the application. These results are summarized in Table 4.3.

ID	Branches	Duration
JOD1	24	25,223 s
VM1	n/a	n/a
MAT1	14	1901 s
COD1	51	496 s
JOD2	28	25,341 s
COD2	17	512 s
JOD3	26	25,542 s

Table 4.3: Number of branches reported by statistical debugging

The first thing to note is that statistical debugging takes more time. However, this time is due to the supplied test cases being set up to generically detect errors, in contrast to the test suites generated by BUGEX to specifically target the bug at hand. In practice, the automated test suite would be run in regular intervals anyway, and statistical debugging would incur only a small overhead on these runs.

The more important difference is the number of branches that statistical debugging can isolate. In all seven cases, the number is considerably larger— from a factor of 1.73 in JOD3, the Parse French Date bug, to a factor of 51 in COD1, the Base64 Decoder bug.

We also observed that the differences of the correlation of the branches to the failure is much lower, rendering the cutoff of the list where the difference is an order of magnitude much more arbitrary.

In all seven defects examined, BUGEX focuses on a far smaller number of branches than statistical approaches.

Usually, statistical debugging approaches are compared by considering an imaginary developer which already has perfect bug understanding, such that he can identify the (seeded) single line of faulty code on sight, without any further contextual information. The higher the faulty line got ranked, the better the approach obviously is. There are cases in which BUGEX also returns only a single line of code. In these cases it would be tempting to ask which rank that line got by the statistical debugging approach. But since for none of the bugs we consider here, the fix consists in a single line of code, we do not consider our result to be the correct answer that we would request from a statistical debugging approach. Besides, since we *only* return a single line of code, the statistical debugging approach would be worse *by design of the comparison*. What we really propose with this way of evaluation is a change in paradigm: Instead of assuming perfect bug understanding, we propose to help developers understand a failure.

4.2.6 Qualitative Results

Let us now discuss RQ2 by assessing whether and to what extent the results help developers in understanding the failure. Since this cannot be verified automatically, we performed a manual investigation, and discuss each failure from

Table 4.1 in detail. We start with the BUGEX report and relate it to the defect in the code.

JodaTime Brazilian Date Bug

This is the failure discussed in Section 1.1. Bug report 2487417 for JODATIME [62] manifests itself with an exception thrown in the code in Figure 1.3.

Branches. BUGEX’s result (Figure 4.11) shows that the failure is due to the mapping of the internal and local time being inconsistent, a condition which is only true (as the comment above the branch shows) if we are near a daylight savings time boundary.

How does this branch lead to the defect? It turns out there is no simple defect in the code; the actual fix involves a redesign of the API with deprecation of several classes and methods and creation of new classes and methods to replace them. Suggesting new code to write is beyond the capabilities of any automated debugging tool; however, BUGEX was able to pinpoint the failure condition to be addressed by the new code. This case raises an interesting point: State-of-the-art approaches to defect localization try to identify defect locations in the code, which makes little sense if a large-scale refactoring and extension is required. By focusing on execution features instead, BUGEX can better guide such refactorings.

Predicates. Since the isolated branch already pinpoints the failure, there is no need to explore the predicates.

Vending Machine Bug

This is a small artificial example of a vending machine that serves as a proof-of-concept, and that we already used in earlier studies on automated debugging [14]. It consists of two classes: the class `VendingMachine` is where the simple business logic is located; whereas the class `Coin` is a mere enumeration of possible inputs. The machine only handles a single price, and thus vending is enabled or disabled depending on the remaining amount of credit after insertion or retrieval of coins and after vending.

Branches. BUGEX returns a single problematic if-clause (Figure 4.18).

```

1     if (this.currValue == 0) {
2         this.enabled = false;
3     }

```

Figure 4.18: Branch returned by BUGEX for the Vending Machine example

The vending machine fails when the branch in Figure 4.18 is not taken, that is, `this.currValue` is non-zero. This causes the machine to stay in `enabled`

state, allowing for a second vending operation, which will bring the credit below zero and cause an exception. This point is precisely the point where the defect is, which means that BUGEX perfectly pinpoints the failure cause in this case.

Predicates. Since the branch already pinpoints the failure cause, there is no need to explore predicates.

Commons Math Sparse Iterator Bug

Apache Commons Math is a library of lightweight, self-contained mathematics and statistics components. Defect number 367 [83] is revealed by the test shown in Figure 4.19.

```

1 public void test() {
2     double[] vdata = { 0.0, 1.0, 0.0 };
3     RealVector vector = new ArrayRealVector(vdata);
4     Iterator<RealVector.Entry> iter =
5         vector.sparseIterator();
6     iter.next().getValue();
7     iter.next().getValue(); // throws exception
8 }

```

Figure 4.19: Test to trigger the Sparse Iterator Bug

In this test, *iter* is a *sparse iterator*, which should iterate over the non-zero values in *vdata*. The second call to *next()* throws a *NullPointerException*.

Branches. For this failure, BUGEX reports eight related branches. All these branches refer to two references in the iterator, namely *current* and *next*, and either compare them against *null* or check the *index* field of the referenced *Entry* against value *-1*. It turns out that the code uses *both these concepts* (being *null* or having an *Entry* of *-1*) to indicate a *non-existing entry*, which causes the failure when their values are inconsistent. The official fix uses only the *-1* marker to detect non-existing entries, thus eliminating the inconsistency.

This example demonstrates that BUGEX minimizes the number of branches to the absolute minimum—but not further. If the failure can be reached through different paths, BUGEX can report them all (as long as the underlying test generation technique is able to exercise those alternative paths).

Predicates. Because the results from BUGEX are not immediately conclusive, for this failure, the developer may chose to examine the reported failure-related predicates. The predicates with the topmost ranking are *index == 1* (again referring to an iterator attribute), *hasNext() == true*, and a different formulation of the first predicate: *getIndex() == 1*. In this setting, *index == 1* means that the iterator already is at the last non-zero element; yet, *hasNext()* is true, suggesting that there would be more elements. This

pinpoints the inconsistency in the iterator, which is the cause of the failure of the call to `next()`.

The reason why BUGEX does not report `current == null` as a failure predicate is that `current == null` is a valid state that frequently occurs in passing generated tests. Hence, `current == null` is a necessary, but not sufficient, condition for the failure.

Commons Codec Base64 Decoder Bug

The *Apache Commons Codec library* provides implementations of common encoders and decoders such as Base64, Hex, Phonetic and URLs. Bug report 98 (see [Figure 4.20](#)) provides predicates under which the failure arises.

Certain (malformed?) input to `Base64InputStream` causes a `NullPointerException` in `Base64.decode()`. The exception occurs when `Base64.decode()` is entered with the following conditions:

- *[The field] `buffer` is `null`.*
- *[The field] `modulus` is 3 from a previous entry.*
- *[The parameter] `inAvail` is `-1` because `Base64InputStream.read()` reaches EOF on line 150.*

Under these conditions, `Base64.decode()` reaches line 581 with `buffer` still `null` and throws a `NullPointerException`.

Figure 4.20: Bug report of the Base64 Decoder bug [21]

Branches. BUGEX reports a single branch as the culprit: `buffer != null` in `Base64.readResults()` (see [Figure 4.21](#)).

```

1     int readResults(...) {
2         if (buffer != null) {
3             // ...
4             if (...) {
5                 // ...
6                 buffer = null;
7             }
8             // ...
9         }
10        return eof ? -1 : 0;
11    }

```

Figure 4.21: Code excerpt for the Base64 Decoder bug

This method and method `decode()`, in which the exception is raised, are called in turns several times in every run. The method `decode()` fills an internal buffer with decoded input, while the method `readResults()` empties that buffer and copies the results into the target array.

For all *passing* runs, in the last call to `readResults()` the `buffer` is already `null`, `eof` is `true`, and thus `-1` is returned. For all *failing* runs, conversely, in the last call `buffer` is set to `null`, and the number of bytes read is returned. The developers decided to fix the issue elsewhere in the code, closely to where the exception was raised. Although this is a possible fix, based on our analysis we believe that the code reported by BUGEX is the actual cause, and not just a symptom, of the failure, and we would have fixed the problem there.

Predicates. While the branch isolated by BUGEX is sufficient to understand and fix the failure, the isolated predicates provide additional information on the failure conditions. The top-ranked predicate is `modulus == 3`, as described in the bug report. The other conditions listed in the bug report are not reported by BUGEX because it can generate passing runs where each of these conditions hold. Therefore, in this case, not only can BUGEX provide additional details over the facts stated in the bug report; it can also show that some reported facts are irrelevant.

JodaTime Western Hemisphere Bug

JODATIME bug report 2889499 [63] is given in [Figure 4.22](#):

```
DateTimeZoneBuilder# toDateTimeZone(String, boolean) creates Period in
PeriodType.yearMonthDay() for inner purposes. This period is created using
DateTimeZone.getDefault() and started on java's beginning of the times.
Period fields are calculated as difference of (minuendInstant + offset) and
(subtrahendInstant + offset). offset is a default zone offset for
subtrahendInstant, and is negative on Western semi-sphere. But
subtrahendInstant is already minimal possible value and can't be increased by
negative value without arithmetic overflow, which was converted to
ArithmeticException in ZonedChronology.ZonedDurationField#
getOffsetToAdd(long).
```

Figure 4.22: Report of the Western Hemisphere bug

Branches. For this bug, BUGEX identifies seven branches that are highly correlated to the failure. Six of the seven branches are related to initializations and set fields to default values if no specific values are specified. BUGEX reports these branches since the failure requires these default values to be used. The seventh branch is where the actual fix was applied.

This result shows that, as expected, BUGEX cannot distinguish between *conditions* for the failure to occur and *errors* that need to be fixed. Nevertheless, the number of branches reported is still low enough that a developer could inspect them to find the one actually responsible for the failure. Moreover, the number of state properties is also small and can help understand the defect.

Predicates. BUGEX reports nine different predicates on five methods. Among these predicates is the very fact that is given in the bug report as the reason for the failure: `subtrahendInstant is already minimal possible value`. Another predicate points to the fact that the error occurs on the calculation of the year

value. The remaining seven facts are artifacts of the test generation process, such as the number of transitions used in the test case. Currently BUGEX cannot find that `offset` is always negative as stated in the bug description, because predicates are only checked on method entry, and not within method bodies or at method exit—where this value is calculated.

Commons Codec Base64 Lookup Bug

Issue number 22 in the apache commons codec library [20] is a failure due to an `ArrayIndexOutOfBoundsException`.

Branches. When we apply BUGEX to this test case, it returns a single branch as the result: the branch from the loop header of the `for`-loop that iterates over the input. When we examine the code further, it shows that the (integer) input is used directly to access a lookup-table, which of course fails whenever the input is negative. There is no branch in the program that captures this critical property of the input. What BUGEX has isolated is that the failure occurs only if the loop body is taken at least once—without input, there is no failure.

Predicates. While execution of the `for` branch is necessary for the failure to occur, it is by no means sufficient. The `for` loop is entered also by passing runs (but with much lower probability). Leveraging its generated test cases, BUGEX isolates two predicates related to the failure:

- The first predicate is `arrayOctest.length() == 3`. This is an artifact that stems from the fact that EVOSUITE did not alter the length of the input array.
- The second predicate is `octect <= 0`. The value of `octect` depends on the input and is used directly to access a lookup table, an operation that fails whenever the input is negative.

Considering also the predicates, BUGEX provides enough information to capture the failure condition—that it suffices to have one negative number in the (non-empty) input for the program to fail.

JODATIME Parse French Date Bug

JODATIME's bug report 1788282 says that parsing a valid french date fails with an `IllegalArgumentException` [61].

Branches. For this issue, BUGEX produces 16 relevant branches. Upon inspection, we found that all branches are necessary for the failure to occur. However, none of them provides a direct explanation of the problem.

Predicates. BUGEX is unable in this case to identify sensible predicates that can explain the failure. The main reason is that, for this failure, the number of predicates is too large and causes BUGEX's analysis to timeout or run out of memory.

The reason why BUGEX does not work for this example is that it uses the exception as the failure detector. That is, we assume that the defect is triggered whenever the exception is being raised. For many of the executions BUGEX generates, however, the exception is raised for actually incorrect inputs, which is the right behavior. Therefore, to be able to successfully apply BUGEX to the given example, we would have to use a more accurate *oracle*.

This is a manifestation of the well known *oracle problem* (see [Section 2.1](#)), a general issue that affects many software testing activities. From a methodological point of view, it would clearly be highly desirable to have formal specifications for the code that would allow for creating perfectly accurate oracles. Unfortunately, this is rarely the case, and the absence of specifications can hurt not only implementation and documentation, but also testing, verification, and (as we see) automated debugging.

Threads to validity

Threats to *construct validity* have to do with how we measured the performance of our debugging technique, by assuming that the number of branches that need to be analyzed is directly correlated with the effort needed for this examination. However, this assumes that branches are suitable to explain faults, whereas in practice different or additional information might be needed.

Threats to *internal validity* might come from how the study was performed. To reduce the probability of defects in our framework, we carefully tested it. To counter the issue of randomized algorithms being affected by chance, we ran each experiment multiple times; the results were the same for each run. The running time is the average over all runs.

Threats to *external validity* concern the generalization to software and faults other than the ones we studied, which is common for any empirical analysis. Our sample size is small; only seven different programs and bugs were used in the study. The reason for this is that it is time consuming to find and reproduce real bugs by manually analyzing bug reports. This produces a bias towards well documented and easy to reproduce issues. However, the set of subjects used represents the entire set of problems BUGEX was used on, and by choosing different subjects, rather than applying BUGEX to many issues on the same subject, we increase the heterogeneity of the sample set.

There are many parameters in BUGEX and the underlying techniques (i.e., EVOSUITE) that we needed to define, such as weights, timeouts, thresholds, and so on. Where applicable, we picked parameters as used in other studies; still, other choices could possibly affect our results [6]. Given that we had close to optimal results in six out of seven cases, however, we believe that changing the parameters could affect the time it takes to search for these results, but not necessarily their final quality.

Summary

After examining the BUGEX results for the seven failures in [Table 4.1](#), we can draw two main conclusions. First, the facts reported by BUGEX provided immediate help in pinpointing the bug in six out of seven cases: either the single branch reported directly led to the defect, or the additional state predicates highlighted important conditions for the failure to occur.

For six out of the seven failures considered, the facts reported by BUGEX effectively led to the failure cause.

Second, the analysis of the seven failures provide initial, but clear, evidence that much of the research in automated debugging has been misguided in pointing to quantitative results alone (“5% of the code”), without actually investigating the qualitative value of the approaches. Our discussion of actual defects shows that, as also discussed by Parnin and Orso [94], helping debugging tasks involves more than producing a list of source code lines—in particular, the whole concept of locating a defect in the code becomes questionable if the fix requires refactorings and extensions.

Failure-related facts, as produced by BUGEX, can provide effective assistance in isolating and understanding defects.

Statistical Debugging

To have a baseline for our results, we also examined the topmost ranked branches reported by statistical debugging ([Table 4.3](#)). We performed statistical debugging using the test suites supplied with the programs considered. The assumption in statistical debugging is that developers would process the ranked list of facts one by one, in order; more realistically, we assumed that developers would do so for at most ten unhelpful diagnoses (see Parnin and Orso [94] for supporting evidence). In all but one case, the top ten branches did not contain the branches reported by BUGEX, nor would they have been as helpful; the only exception is the *Commons Math Iterator bug*, where the top ten branches reported by statistical debugging contained many of the branches reported by BUGEX.

Statistical debugging using existing test suites does not produce as helpful results as BUGEX.

We conclude that statistical debugging works best when used in conjunction with a tailored test suite, as also observed by Artzi et al. [7]. However, if such a test suite is to be generated, one may just as well guide its generation based on the bug at hand—which is precisely what BUGEX does.

The results also show that, by focusing the generation of additional executions on the failure at hand (i.e., by employing b_{last}) we were able to overcome the limitations of the first approach ([Section 4.1.4](#)).

4.2.7 Limitations

Failing test case

As explained in [Section 4.2.2](#), input to our approach is a program p to examine as well as a failing test case t_{fail} . Currently, this test case needs to fail with an exception rather than with a failing assertion. The reason being that since we generate additional test cases, we cannot automatically determine how the existing assertion suits the new test cases to distinguish passing from failing runs. If for instance the output to a XML API is checked for general properties like “has 5 child nodes”, this might well generalize to the new tests. However if the check is more dependent on the actual input like “element has name ‘marvin’ and id ‘5’” then it might not generalize to new test cases as well. But since this can’t be decided automatically, we need to drop assertions altogether, running into the well-known “missing-oracle”-problem for generating test cases.

Wrong Failure Predicate

Our approach uses the occurrence of the failure as the failure predicate. So we assume that it is *always* incorrect behavior that the failure occurs. However, as was illustrated by the JODATIME Parse French Date bug ([Section 4.2.6](#)) in the case study, there are situations in which failing is the *correct* behavior.

Reproducing the failure

Our approach needs to be able to manipulate the circumstances that are crucial to trigger the failure. If for instance the tests makes use of an existing file that is used as input, or the failure is non-deterministic, our approach will fail.

Limitations of the underlying technology

The tool relies on the employed test generation technique to come up with test cases that execute b_{last} . These techniques currently suffer from a number of problems in that regard. Where dynamic symbolic execution techniques suffer from difficult to solve constraints, such as hashing- or cryptographic functions, search based approaches have problems with structured string input such as XML.

4.3 User Study

Parts of the contents of this section have been published in Rößler [101].

We claim that the results produced by BUGEX will help a developer to understand a failure. And we showed seven real-life examples in detail to underpin our claim. However, these seven examples are too small a set to ensure that

the claim generalizes to other failures as well. To fill that gap, we prepared and started to conduct a broad user study with 373 students. However, because of technical hardships imposed by our prototypical implementation, we were unable to complete the study. We want to present the results of our efforts nonetheless for other researchers to be able to draw from our ideas as well as learn from our mistakes.

4.3.1 Research Questions and Hypotheses

First we will present the research questions we tried to answer with the study in the form of hypothesis to falsify. As mentioned earlier, we deem BUGEX helpful for developers to understand a defect. Thus our first research hypothesis (RH) was:

RH1: BUGEX is valuable for the debugging task.

This is a purely subjective evaluation criterion that indicates whether users like to use the tool. We consider our tool to be valuable for users, if they continuously prefer to use the tool over not using it. Additionally, we would ask the users after each bug they fixed, whether they liked the tool, what they liked about it and what could be improved.

Our approach belongs to the field of automated debugging and has to compete with other approaches from that field:

RH2: BUGEX is more helpful than state-of-the-art statistical debugging approaches.

We would consider our tool to be more helpful than an alternative approach if users continuously prefer it over the alternative. Additionally, we would request the estimated amount of time needed for fixing a bug, how helpful users considered the tool and how confident they are in the fix they implemented.

If a defect is expressed only in terms of a failing test case, then there are many equally valid possibilities to fix it. The question is, whether developers are aware of this:

RH3: Developers are aware that there are multiple equally valid ways to fix a defect.

To test this hypothesis, we would ask users after each bug they fixed, whether they had alternatives and how many.

We assume that different fixes have different probabilities to be chosen by the developer. Furthermore, we assume the decision for a specific fix (including a rewrite of the complete program) is influenced or even mainly driven by the expected overall effort, which includes foreseeable future changes and fixes to probable future bugs. To make a concrete example: if the developer thinks,

that implementing the program with a different design will save some effort in the long run, and if there are no additional factors such as upcoming program release dates, then this will be the fix of choice.

RH4: *The main reason to chose a specific fix is that it bears the least amount of estimated overall effort.*

To test this hypothesis, we will ask users after each fix for which they were aware of alternatives to fix it, which factors influenced their decision for the fix they eventually chose.

Many studies (including the one by Parnin and Orso [94]) show that users will only consider a few results before giving up on a tool. So we assume that less results are more valuable, even if they do not contain the optimal solution.

RH5: *Users prefer a smaller result set over a larger one—even if it does not contain the optimal solution.*

We consider this hypothesis confirmed if users continuously prefer a smaller result set over a larger one.

4.3.2 Design of the User Study

Typical problems of user studies with developers include the following:

1. The study is performed with only a few developers or does not exceed a certain time boundary (or both).
2. Either developers work on different tasks and thus the results are not comparable, or developers are all given the same program, which encompasses that
 - (a) all programmers are alien to the code and
 - (b) the code cannot exceed a certain size and complexity (see also 1).

All of this limits the generalizability of a study. To address these problems we wanted to perform a different user study. We utilized the participants of a course in programming that is due in the second semester of studying computer science at the Saarland University. The course is held in German, so all surveys were in German as well. 373 students of vastly differing programming experience registered for the course, but only about 154 successfully completed it.

During the course, all of these students were given four programming tasks with increasing levels of difficulty. Starting with a simple *Hello World* program, the students implemented *seam carving* and a *satisfiability solver*, and eventually ended with a *compiler* of a small artificial programming language. The students had a certain amount of time (1–4 weeks per project) to complete the programming tasks. That amount of time increased with the difficulty of the

tasks. They worked unmonitored on their chosen workstation, in their chosen environment and at their chosen time. They could suspend and continue to work any time.

All intermediate programming results were submitted to a central server and built and tested every night. The students received feedback from automated unit tests. If all tests passed, the programming task was assumed to be completed. Some of the tests were given to the students together with the task and could be run on the workstations as often as needed and were also run on the server after every commit. Some other tests were unknown to the students—they were executed *only* on the server and only once every night (called *nightly* tests). Of these nightly tests, students only received the failure message if a test failed. This created two different usage scenarios: a usage scenario that resembles normal development, where tests can be executed and manually debugged as often as needed, and a scenario which resembled that the code was executed on end user’s machines, where students cannot rely on manual debugging at all (without using RECORE that is).

For all failing tests (after every commit for the given tests and every night for the nightly tests), our BUGEX prototype would generate *diagnoses* (i.e., branches and state predicates highly correlated to the failure) that the students could request via a web interface. Using this feedback was optional. We detected every request of the diagnoses, and thus were able to see how beneficial the students deemed them to be (i.e., if they did not cease to request them). When requesting a diagnosis, the students were required to complete a very short survey before being granted access. Since we assumed that students might have been annoyed by the questions if they were too much effort to fill out, all of the questions were optimized to be answered quickly.

Befragung zu SVN-Revision 1234 vom 01.05.2012

Achtung: Seit Sie das letzte Mal diesen Fragebogen beantwortet haben, haben Sie 2 mal eingecheckt. Dieser Fragebogen bezieht sich jedoch nur auf die letzte Änderung vom 01.05.2012 mit Revisionsnummer 1234. [SVN-Diff der Änderung](#)

Ihre Checkin-Nachricht war:

1. Welche Änderungen, die Sie mit dieser Revision eingecheckt haben, haben Sie am Projekt vorgenommen?

Fehler behoben

Funktionalität hinzugefügt

Refaktorisierung

Andere Änderungen:

Optionale Anmerkung

Figure 4.23: Survey on the last committed revision

If they had committed new revisions since they requested the last diagnosis, they were asked about the last committed revision (Figure 4.23). To refresh their memory, some details about the revision were given, such as date and time, revision number, commit message and a link to view a diff. They were asked which changes to the project were committed with that revision. Possible answers were “fixed defect”, “added functionality”, “refactoring”, and “other” with a text field. This question would allow us to disregard changes that were

not fixes. Multiple answers were possible and the question had an optional field for comments.

2. Sie haben gerade angegeben, dass Sie in dieser Revision Fehler behoben haben. Um welche Fehler handelt es sich?

prog2.project1.MyClassTest#testMethod

prog2.project1.MyClassTest#testOtherMethod

Anderer Fehler:

Optionale Anmerkung

Figure 4.24: Additional question on the last committed revision

If the students selected “fixed defect”, a second question appeared (Figure 4.24). Students were asked whether they had fixed a defect that had made a test fail previously, and if so, which one (multiple answers were possible). All previously failing tests were listed together with a “other defect” text field. This question would allow us to associate the survey feedback with a certain failing test and diagnosis, so that we could reviewed later which results of the automated debugging tools were more helpful. The question had an optional field for comments.

Befragung zu Fehler im Test #

Dieser Fragebogen bezieht sich ausschließlich auf den Fehler #, von dem Sie angegeben habe, dass Sie ihn in der Revision vom behoben haben.

1. Wie lange hat es gedauert den Fehler zu beheben?

ca. Minuten ich hatte Hilfe von Dritten

Optionale Anmerkung

2. Wie zuversichtlich sind Sie, dass Ihre Änderung den Fehler behebt?

behebt den Fehler sicher

behebt wahrscheinlich den Fehler

bin nicht sicher, ob der Fehler behoben wurde

behebt den Fehler sicher nicht (Workaround)

kann es nicht einschätzen

Optionale Anmerkung

3. Warum haben Sie sich für diese Möglichkeit entschieden um den Fehler zu beheben? (Wenn Sie mehrere Gründe hatten, nennen Sie den wichtigsten.)

war die einzige Möglichkeit

ging schnell / war einfach

war sauber / elegant

anderer Grund:

Optionale Anmerkung

Figure 4.25: Question on the fix committed in the last revision

If students indicated they had fixed at least one of the listed defects, the survey continued with another set of 3–4 questions (Figure 4.25), concerned with *one* of the fixed defects, identified by the previously failing test. If students selected more than one defect, one of the failing tests was chosen randomly. The first question asked was how long it took them approximately to fix the defect (in minutes) and whether they had help. The second questions asked was how confident they were in their changes (because they could not rerun the nightly tests locally). Possible answers were “Certainly fixes the defect”, “Probably fixes the defect”, “Not sure, whether the defect is fixed”, “Doesn’t fix the defect (workaround)” and “Don’t know”. These two questions would help us to answer **RH1** and **RH2**. The third question asked why they decided for this way to fix the defect. Possible answers were “was the only possibility”, “was fast / easy”,

“was clean / elegant”, and a “other reason” text field. This question would help us answer **RH4**. All questions had an optional fields for comments.

Figure 4.26: Additional question on the fix committed in the last revision

If student selected anything but “was the only possibility” in the third question, a fourth question appeared (Figure 4.26), asking how many other possibilities they had recognized. That question also had an optional field for comments. This question targeted at answering **RH3**.

Figure 4.27: Optional questions on the fix committed in the last revision

At the bottom of the survey, there were three optional free-text questions (Figure 4.27). The first question asked them to describe the problem in their own words. The second question asked them to describe the fix in their own words. The third questions asked how an ideal explanation of the problem would have looked like. The text field for that questions contained some example answers: “location in the code / property of the execution / ...”. And at the end, there was another text field for additional remarks.

Figure 4.28: Survey on the last diagnosis the student received

If students had viewed a diagnosis before (no matter whether of the same or of a different revision), they were asked a single question (Figure 4.28): “How helpful was the last given diagnosis to understand the defect?” There were three possible answers: “very helpful”, “little helpful”, “not helpful”, “didn’t use diagnosis”. The question had an optional field for comments and a link to

the diagnosis in question, in case the student was not sure which one that was. This question was targeted to check **RH1** and **RH2**.

Completing the surveys was only enforced by not giving access to further diagnoses. So if students would cease to use the feedback, they also would cease to complete the surveys. However, as some of the tests were only executed on the server, we expected students to be interested in getting as much additional information about these nightly tests as possible. Therefore, even if students would have ceased to use the feedback of the given tests in favor of conventional manual debugging, they still would have wanted to receive the feedback for the nightly tests.

This also bore another advantage: students might have been reluctant to try and become acquainted with an alternative debugging method. But if they would have continued to use the feedback for the nightly tests, this might have helped to overcome this inhibition and would have familiarized them with the feedback from the system. As a downside, since the complexity of the tasks also increased over time, this would have masked whether automated debugging is more helpful for more complex debugging tasks.

The approach addressed all of the typical problems a user study faces: The study was performed with a large group of developers and there was practically no boundary on the time the developers could take to fulfill the programming task. Also, the students would work with *their own* code. This meant that they already knew the code sufficiently well and that the code could be of arbitrary size and complexity (but, of course, directed towards solving the given problem). Still the results would have been comparable, as all participants worked on the *same tasks*. The tasks were mandatory to pass the course, so the students had a strong motivation to complete them. And because they could not use conventional manual debugging on the nightly tests, they had a strong *intrinsic* motivation to use the alternative debugging approach.

To appropriately address the research questions formulated in Section 4.3.1, we offered three different options for diagnosis each participant could choose to receive. After choosing an option and before being able to receive the feedback of the next option, the participant had to complete the short survey as shown above (Figure 4.28). Additional to the survey results, for each option we would continuously capture the number of times users requested it. The three options are:

- **OPT1** This option was the feedback of a state-of-the-art statistical debugging approach and established the ground truth that would have allowed us to check **RH2** and **RH5**.
- **OPT2** This option was the feedback of a state-of-the-art statistical debugging approach, but the feedback only contained a limited number of results. This option allowed us to check **RH5**.
- **OPT3** This option was the feedback from BUGEX that would have been compared to **OPT2** to check **RH1** and **RH2**.

The options were assigned neutral names and the assignment of names to options differed for each student.

Threats to Validity

We identified several possible threats inherent to the design of the study.

This user study was conducted with students only, which introduces a sampling bias. A short mandatory survey at the beginning of the course about the programming skills and experience of the participating students showed that 341 students had no practical programming experience. 24 students had practical experience ranging from half a year to 3 years. Only 8 students had programming experience of 4 years or more.

One problem that may have arisen with the design of the study was the introduction of a self-selection bias: by selecting an option and respectively a system students wanted to receive feedback from, they select themselves into a group. We wanted to counter this thread by the following measures: The assignment of the system that generates the feedback to the neutral option name was kept secret, and different for every student.

Another risk came from the *Hawthorne-Effect*. This effect describes the phenomenon, that participants of an intervention study in worklife will behave differently simply because they know that they participate in a study. However, the very existence of this effect has been questioned recently [116], and the results of the corresponding studies have been attributed to other unmonitored factors. Generally, it is impossible to conduct studies with human beings where all possible factors are controlled or even monitored. Observing effects that originate from factors that have been (intentionally or unintentionally) ignored is a major thread to every study on human beings.

4.3.3 How the Study Failed

When preparing the study, it was not yet clear which projects the students would work on. In earlier instantiations of this course, the students had projects where they implemented a red-black-tree, a spell-checker, the burrows-wheeler transformation and games such as tetris and pac-man.

The final decision for the projects was only made shortly before the start of the course (or even shortly before the start of the project). So there was little time to prepare BUGEX and the underlying infrastructure. The final set of projects which was chosen was a little unfortunate for the means of the study:

The first project was *Hello World*, as usual. This project does not contain much code and really only serves as an easy starter, to familiarize the students with the process and the tools. As a result, there were only few failing tests.

The second project was *seam carving*. That project would have been a good match for BUGEX. However, as shown in Section 4.2, BUGEX needs to transform the existing failing test case into some internal format, that EVOSUITE can apply the genetic algorithm on. Former tests were simple method sequences, very similar to the internal format of EVOSUITE. But the tests for this project consisted of complex code, containing `for` and `while` loops and `if-then-else`

statements. And since we only got to know that shortly before the project started, there was only little time to add the functionality to BUGEX that let it transform such code into the internal format. When we were done implementing this (by executing the code and recording which statements got executed, thus unrolling loops and dissolving `if-then-else` statements), the project was already completed by the majority of the students.

The third project was a *satisfiability solver* and the fourth project was a *compiler*. These projects were mostly working with `String` input, where that input was expected to have some complicated and delicate internal structure. That is for the satisfiability solver we would have had to generate valid boolean formulae and for the compiler, we would have had to generate valid source code. Both of this exceeds the possibilities of any currently known general execution generation technique, and thus also exceeded the possibilities of EVOSUITE. Eventually this forced us to abort the study.

The course this study utilized takes place only every other semester. So the next possibility to conduct the same study with the same course is only after this thesis will be already submitted.

4.3.4 Conclusions

To the best of our knowledge, the user study as proposed in this paper is unprecedented in the field of automated debugging. This made it very promising in its ability to counter issues we detected in earlier studies. But at the same time, this caused many uncertainties and risks due to the lack of experience with such a study. And as eventually turned out, such an uncertainty (i.e., which projects would be chosen and what the automated tests would look like) eventually forced us to abort the study.

We did not want to run BUGEX on a small set of well-prepared failures, and then present the result of these to a small number of participants of the study. This would have been much safer, but would have come with the cost of the before mentioned problems (foreign code, unrealistic work environment, limited amount of time and participants). Instead we opted to run BUGEX “in the wild”, i.e., with unknown failures, unknown test cases and unknown target programs. The result of this showed the limits of BUGEX as it is only a prototype, missing many features and needing much more testing before being ready as a robust industry-scale product.

This study showed how much effort the design and preparation of a user study requires, and how easy error prone it is. This is probably the reason, why there are still so few user studies in the field of automated debugging.

Chapter 5

Conclusion and Future Work

In the following we will review the achievements and contributions of this work to the field of automated debugging and outline what is left to be done in the future.

5.1 Conclusion

Debugging is a time-consuming, tedious task and a major cost factor in software development. Of course the obvious solution is to omit debugging altogether by not making mistakes in the first place. However, despite all efforts and current research, we assume software will continue to contain defects in the near future. So we need techniques that help us to debug software.

The debugging process consists of four steps. As already pointed out, the *registration* of a failure is already automated. In today's end user applications, the user can send an automatically generated bug report with the push of a button.

The next step is to *reproduce* the failure. So far, the developer needed to manually create an automated test case that lets them replay the failure at will for investigation and validation of the fix. We automated this step and are able to generate such an automated, self-contained test case with nothing more but the core dump.

The next step in the debugging process is to *understand* how the failure comes into existence. We proposed a general debugging framework that can pinpoint the failure in terms of arbitrary runtime information by applying the scientific method.

The last step in the debugging process—fixing the defect—can arguably never be fully automated for arbitrary failures; otherwise one could define the defect to be the absence of the program and get the code created automatically.

Therefore, our tool chain automates as much as possible, saving a lot of effort and reducing overall software development and maintenance cost.

In detail, the contributions of this thesis are the following:

RECORE Our proposed approach generates a minimal, self-contained test case that recreates a failure from the core dump. It should be emphasized that this approach does not merely replay the failure by loading objects with possibly already faulty state into memory, which would not allow developers to draw conclusions on how the infection came into existence. Instead we start from a sane state and create the faulty state and ultimately trigger the failure by a sequence of method calls.

Our implementation uses a genetic algorithm that is seeded with the values as found on the core dump, and a custom fitness function that optimizes the test towards similarity with the failure and eventually reconstructs executions that reproduce the failure. Thus we showed that search-based test case generation is useful not only for exploring unknown behavior, but also for reproducing *known* behavior. Applied on seven JAVA bugs, RECORE was able to exactly reconstruct the failure in five cases, and partially in another case. The RECORE approach incurs zero overhead in production code; the resulting test cases capture the essence of the failure, are easy to understand, and easy to run through a debugger.

BUGEX Our approach is a general debugging framework to show the correlation of arbitrary runtime information (called *facts*) by employing the scientific method (observation, hypothesis, experiment, refinement). First we record the facts for a failing test (observation) and create an initial correlation of those facts to the failure (hypothesis). In an iterative manner, we refine that correlation by creating additional executions that challenge it (experiment and refinement). The result is a correlation of unprecedented accuracy.

This general framework can be implemented for arbitrary execution features and use any execution generation technique, targeting a certain such fact. We implemented it for branches and state predicates, using a genetic algorithm as execution generator. We apply state-of-the-art statistical debugging methods to calculate the correlation of individual facts to the failure. By systematically generating test cases, BUGEX can isolate facts that precisely characterize when and how the failure occurs. Unlike traditional statistical debugging approaches, BUGEX requires only a single failing run, which is the starting point for any debugging activity. The results of our preliminary evaluation of BUGEX are encouraging: in six out of seven cases, the features isolated by BUGEX pinpointed the failure cause.

5.2 Future Work

Despite the advances shown in this work, there is still much to do. The next step we are already working on, but which will not be completed before this thesis due to the sheer amount of computation time needed, is a much larger evaluation.

Besides general improvements such as scalability, stability, and speed, there are some research questions worthwhile to investigate upon in regard to both RECORE and BUGEX.

5.2.1 Large-Scale Evaluation

The main concern regarding the results presented in this thesis is the question: how representative is the small sample set used in the evaluation? As described in [Section 4.3](#), we tried to evaluate BUGEX in a large scale user study during a course in programming. However, due to technical difficulties and because the projects for the course were prepared on short notice, the study had to be cancelled.

We also started to prepare a quantitative evaluation of unprecedented scale (see also [Section 2.8](#)). But due to computation time required to prepare that evaluation, it is yet unfinished.

The ground truth in automated debugging is always the fix. But as discussed in [Chapter 1](#), in general, there are infinitely many ways to fix any given defect. Yet somehow, the developer chose one of those fixes above all others. If we could recognize a rhythm and rhyme to that, and extract heuristics to determine in which situation which fix is most appropriate, we would go a long way towards helping a developer understand a failure, suggest possible fixes or even fix a defect fully automatically.

The same applies not only to fixes, but also to bugs. To our knowledge, there is little data on how bugs are introduced during development. If we could recognize some pattern of how such bugs look like, in what situations they are introduced, and what other effects they have, we could improve bug prevention tools and techniques such as PMD [\[95\]](#), FindBugs [\[30\]](#) and Checkstyle [\[16\]](#).

To evaluate existing approaches and tools properly, we need a large set of real-world failures and fixes. With today's large body of freely accessible open-source software as well as public bug trackers, we are in a unique situation: the source code of the programs, the failures and their complete documentation, and the tests that reproduce the failures are all accessible. We just need to extract and refine them. And since there are so many of them, we can even do so in a fully automatic fashion and afford to lose some of them on the way. Which is exactly what we are working at.

In this chapter, we propose a system (called TRUEBUG) to automatically extract a large body of failures and minimal fixes, using existing revision control systems, unit tests and delta debugging. The very same system can be used to

also automatically extract a large body of defects as they were introduced into projects during development. Together, they can form a realistic benchmark of size, quality and diversity previously unheard of, for the evaluation of automated debugging techniques. Additional to that, they can be used to gain new insights into the nature of both defects and fixes.

We already started to work on this. But since we deal with huge amounts of projects and even greater amounts of data, the overall process takes a lot of computation time and has not finished yet. However, we will present some preliminary result to show the feasibility of the approach.

Schematic approach

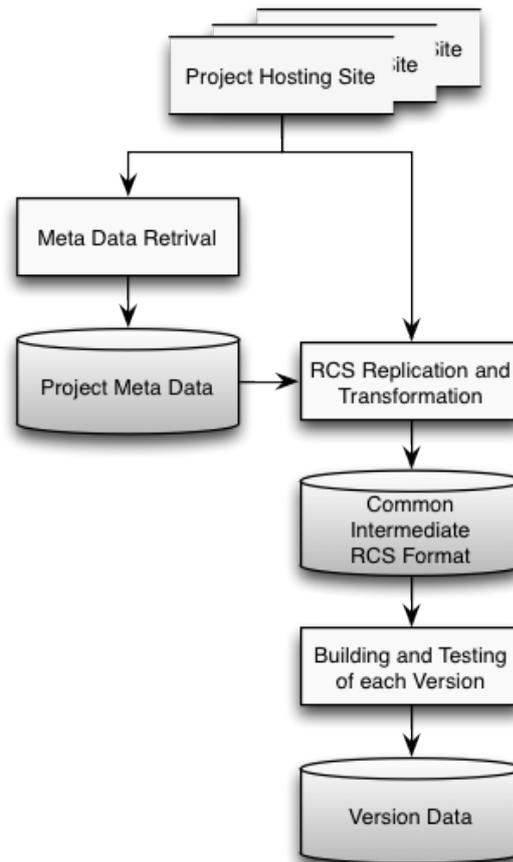


Figure 5.1: A simplified scheme of TRUEBUG

The schematic approach of TRUEBUG is depicted in Figure 5.1. In a first step, the standardized project web site on the project hosting site is used to capture some meta data about the project, such as its name, registration date, number of filed bugs, number of recommendations and weekly downloads. In

a second step, the complete revision history of the project is retrieved from the publicly available source code repository, whose URL is also given on the project website, and transformed into a common intermediate format. This intermediate format is then used to step through the complete history and build and test every version of the project. While doing so, we capture all sorts of data, such as duration and outcome of the build and individual tests. The produced data is saved in a database for later mining.

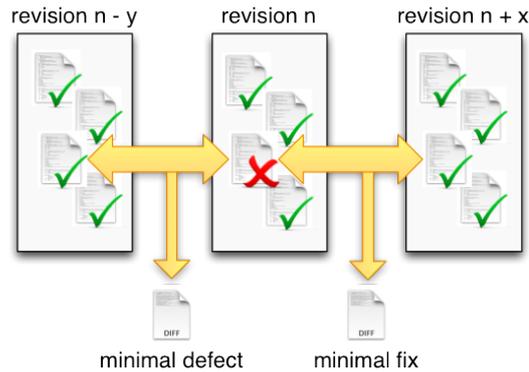


Figure 5.2: TRUEBUG generates a minimal defect and a minimal fix from a failing test by applying delta debugging on the revision differences

Then we iterate over all test cases that we detected to fail. For each such failing test case, we take the first subsequent revision where the test case does not fail. By applying delta debugging between the two revisions and using the test case as oracle, we can extract the 1-minimal fix that makes the test pass (Figure 5.2). Likewise, we can use the last previous revision where the test case did not fail and also apply delta debugging to extract the 1-minimal change that introduced the defect.

What looks like a simple approach in theory is awfully intricate in reality as there are many different combinations of project hosting sites, web site layouts per hosting site, revision control systems, build systems and test systems. And adding to this complexity is the sheer amount of space and bandwidth needed to download and store complete project histories as well as the needed computational power to build and test each and every revision of each project (possibly multiple times when minimizing differences). For each failing test, we extract both minimal defects and minimal fixes by applying delta debugging. This means that there are between $O(\log(n))$ and $O(n^2)$ steps of minimization and test execution (with n being the number of differences). Per failing test, this is done twice—for calculating both the minimal fix and defect.

We found projects where a single revision already was bigger than 5.8 GB. Only the head revisions of the approximately 43,000 JAVA projects we downloaded from SOURCEFORGE summed up to more than 1 TB of files. For big software systems, a duration of a single build of up to 19 hours has been reported [85], where there can be hundreds of thousands of revisions per project. And since there are whole books about how to set up and maintain integration

servers to execute automated builds, how can we hope to fully automate the task of configuring and running builds? And all of this for an overwhelming amount of projects; SOURCEFORGE alone hosts some 430,000 projects [117].

The obvious challenges of this venture are probably the reason why no one else has ever tried to conduct such an analysis before.

Implementation

Our approach makes only some very general assumptions that should be met by most open source project hosting sites. We implemented it first for SOURCEFORGE [108], as to our knowledge, this is currently the biggest open source project hosting site [117].

Although the approach is able to handle all kinds of projects, we initially restricted ourselves to JAVA projects in order to get a more homogenous set of projects to work with. Instead of downloading the complete version history at once, we only checked out the latest version of the project to see whether we could recognize its build system(s). In order to do this, we went through all files and tried to identify the build system by commonly associated file names. For instance, `makefile` would indicate MAKE [81], `build.xml` files indicate ANT [5] and `project.xml` or `pom.xml` point to two different versions of MAVEN [84] as build system. Since our depicted approach depends on the existence of tests to be able to detect and minimize both defects and fixes, we also required the projects to contain at least a single JAVA file with “Test” in the name¹.

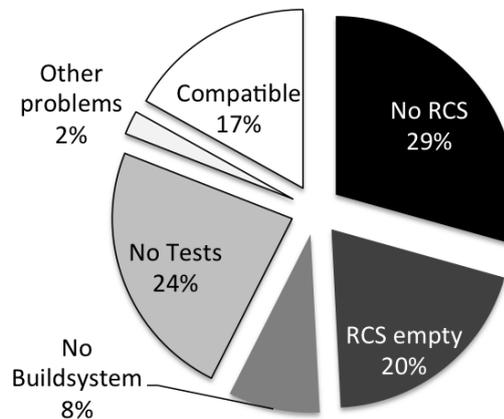


Figure 5.3: Percentage of mined projects compatible with TRUEBUG

We discarded 29% of the projects because there was no URL pointing to a revision control system on their project web site (which can of course also mean that it is hosted elsewhere). Another 20% were discarded because the given revision control system was empty or contained no JAVA files. 8% of the

¹JUnit3 has the mandatory naming convention that the class name has “Test” as postfix, while JUnit4 requires an annotation of a test method with `@Test` [66].

projects did either not contain any build system or we were unable to recognize it. And another 24% of projects did not contain any tests according to the described file pattern. 2% of the projects were discarded for a whole lot of different reasons. Only if we could recognize the build system and find some tests, we marked the project as compatible and continued with our approach. As can be seen in [Figure 5.3](#), from the approximately 43,464 JAVA projects we started with, this already excluded 83% of them, leaving us with 7,328 projects.

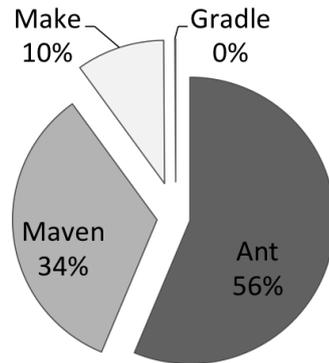


Figure 5.4: Distribution of unique build systems per project

Of all the projects we could analyze the files of (including some that did not have tests), 13,607 had at least one build system we could recognize. For some projects we found no less than 7 different kinds of build systems and up to 54 different instances of those build systems for different subprojects. The distribution of the 19,295 build systems unique per project (that is each build system was counted only once per project, but a project could have several build systems) is shown in [Figure 5.4](#). As can be seen, 10,196 projects contain a `build.xml` file and thus are likely to use ANT [5] as build system (56% of the build systems). 6,112 projects use MAVEN [84] as build system (34% of the build systems), where there are roughly equally many using MAVEN 1 and MAVEN 2. 1,790 projects use MAKE [81] as build system (10% of the build systems). We found other build systems (such as GRADLE [41]) to not have a substantial distribution.

We also considered some other file patterns within the projects, to recognize which Integrated Developments Environments (IDEs) were used. In 12,311 different projects we found artifacts pointing to 13,013 different IDEs unique per project. As can be seen in [Figure 5.5](#), ECLIPSE [28] is pretty dominating, with its artifacts being found in 10,253 projects (80% of the IDEs). We found artifacts of NETBEANS [91] in 2,404 projects (19% of the IDEs). Of JDEVELOPER [58], INTELLIJ IDEA [51], and JCREATOR [57], we only found artifacts in 90, 57, and 31 projects respectively (together 1% of the IDEs). The artifacts of the build system are a natural extension of the code and are therefore expected to get committed to the revision control system. In many projects it is up to the developer to decide which IDE he wants to use, and often different developers use different IDEs within the same project. As a consequence and in contrast to committing artifacts of the build system, is often assumed to be bad

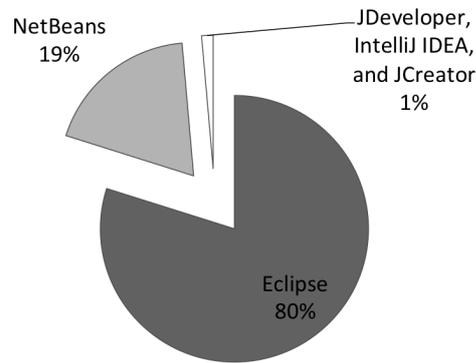


Figure 5.5: Distribution of integrated development environments

practice to commit artifacts of the IDE. Therefore, we did not expect to find that many projects with IDE artifacts. And as another consequence, the results may be biased—for instance, some IDEs might make it harder to configure a project and thus those artifacts are more likely to end up getting committed and shared between developers.

Limitations and Future Work

We are well aware that the numbers presented cannot be considered to generalize to all kinds of projects (i.e., non-JAVA or commercial projects). However, although we only retrieved data from a single project hosting site, we assume that they generalize to open source JAVA projects from other free project hosting sites. We are aware that *free* project hosting sites might attract only a certain kind of open source projects or only projects up to a certain size. And finally, the fact that we simply discard projects that our approach does not work with introduces additional bias.

As indicated earlier, these are preliminary results that merely show the feasibility of our approach. The next step is to download the complete version history and convert the revision control system to MERCURIAL [88] (our common intermediate format). We then execute the configured build systems within the projects for each and every revision. For every failing test we find, we create the 1-minimal difference to the next and previous passing revision, thus extracting both the minimal defect and the minimal fix. We heuristically optimize the process where it seems beneficial. Here the vast amount of projects available comes in handy: we can afford to only support the majority of projects. If a project has some quirks that do not work with our automated approach, we simply discard it (although that introduces some statistical bias).

As mentioned above, we detected to be able to use 7,328 projects from SOURCEFORGE. If we can only build 20% of the projects, this will still leave us with about 1,400 projects to capture build data from. On average, the revision control systems of projects that are compatible have 766 revisions. If we can only successfully build 20% of those revision of the 1,400 projects, we will get

about 214,480 successfully built revisions. If only 20% of these revisions contain one failing test and if we are able to find previously or subsequently passing tests for only 20% of these failing tests, we will still end up with about 8,579 minimal fixes and failures.

We are already working on these steps. However, due to the needed computation time, we decided to only show some preliminary results in this thesis.

We also consider to expand the approach to similar project hosting sites such as `GOOGLE CODE` [40] and `GITHUB` [37] or to project types other than `JAVA` projects. Also, we want to crawl custom project web sites and recognize URLs that point to the respective revision control system to enrich the data and increase its diversity.

5.2.2 Future Work on RECORE

We successfully showed the feasibility to reconstruct failures from core dumps via test case generation. However, our implementation is only a proof-of-concept and can be extended in the following ways:

System test generation. A new generation of test generators [43] applies search-based testing at the system level, synthesizing GUI events as inputs. Such generation techniques could also be integrated into RECORE, resulting in tests that work at the GUI level and mimic user behavior instead of at the unit level.

Capture more events. As illustrated in Section 4.2.6, the lack of history can effectively prevent RECORE from reconstructing the failure. Following the BUGREDUX way and investigating which events can be captured during execution could maximize the effectiveness of reconstructing failures, while minimizing the overhead on executions. For instance, RECORE could exploit events that can be logged at low overhead, such as GUI interaction.

Parallel execution. Multi-threading failures are among the hardest to reproduce and debug. Techniques from execution synthesis [123] could be adapted to control and reconstruct failure-inducing thread schedules.

Improved execution generation approaches. Both RECORE and BUGEX rely on EVOSUITE to generate additional executions. Improving EVOSUITE would thus have a doubly positive effect. One possibility to extend EVOSUITE is to integrate symbolic execution and constraint solving to speed up test case generation in many situations.

User study To fully evaluate the effectiveness of a tool that yields qualitative results (that is results that are not measurable directly), a realistic user study should be performed. Our first attempt to perform such a study turned out to be fruitless (see Section 4.3). However, this merely means that such a study should be performed with a different setting and, most notably, with different projects, that can better be handled by today's execution generation approaches.

5.2.3 Future Work on BUGEX

Despite our initial successes, the combination of test case generation and automated debugging is still in its infancy. Future work will focus on the following topics:

More runtime facts. Besides branches taken and state conditions, there are several other runtime facts that may characterize failures (see [Section 4.2.1](#)).

We are currently exploring the wide range of *test criteria* for this purpose: sub-conditions fulfilled, definition-use relationships, number of loop iterations, and others. We believe that considering a richer set of facts will help with even better diagnoses (e.g., “the failure occurs whenever this loop is taken only once”). One challenge, when considering a new type of facts, is how to define an appropriate fitness functions, as well as how to integrate the findings in a single ranking.

Multiple facts. BUGEX currently only associates individual facts with failures. An obvious extension would be to check for the correlation of *multiple facts* (e.g., “the failure occurs only if the state predicates `current == null` and `hasNext() == true` hold simultaneously”).

Test suites and multiple failures. In case an existing test suite with multiple failing tests relating to the same failure (defined by b_{last}) exists, these tests can be used as seeds for EVOSUITE. We will investigate to what extent this can improve the effectiveness of the approach. This would also allow us to extend BUGEX so that it can search for multiple causes (and their interferences) in parallel, rather than treating each failure individually.

Better diagnoses. Being essentially search-based and nondeterministic both in isolating and test case generation, there is always a chance to improve BUGEX results. One could also focus on an even stronger guidance of test case generation based on test results; we also strive for support for multiple interfering defects [2].

Additional test generation capabilities. Right now, we currently restrict EVOSUITE to only change the values that appear in the given failing test case. However, EVOSUITE is also able to alter the existing series of method calls. There might be cases for which this is beneficial.

Integration with minimization. JINSI [14] minimizes failing executions to a fraction of their size, using a combination of dynamic slicing and delta debugging. We are currently integrating JINSI’s minimization and BUGEX’s isolation capabilities, which should allow us to decrease the search space while further increasing precision.

User studies. So far, studies in debugging have focused far too much on quantitative aspects, widely ignoring the usefulness of the results for developers. Due to the technical limitations of our prototypical implementation of BUGEX, we had to abort the first attempt to perform such a study (see [Section 4.3](#)). But this merely means that such a study should be performed with a different setting and, most notably, with different projects, that can better be handled by today’s execution generation approaches.

Bibliography

- [1] ABREU, R., AND VAN GEMUND, A. J. C. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *Eighth Symposium on Abstraction, Reformulation, and Approximation* (Lake Arrowhead, California, USA, August 2009), SARA '09.
- [2] ABREU, R., ZOETEWELJ, P., AND VAN GEMUND, A. J. C. An observation-based model for fault localization. In *Sixth International Workshop on Dynamic Analysis* (Seattle, Washington, 2008), WODA '08, pp. 64–70.
- [3] AGRAWAL, H., HORGAN, J. R., LONDON, S., AND WONG, W. E. Fault localization using execution slices and dataflow tests. In *ISSRE* (1995), pp. 143–151.
- [4] ALTEKAR, G., ZAMFIR, C., CANDEA, G., AND STOICA, I. Automating the debugging of datacenter applications with ADDA. Tech. Rep. UCB/EECS-2011-22, EECS Department, University of California, Berkeley, Apr 2011.
- [5] ANT. <http://ant.apache.org>.
- [6] ARCURI, A., AND FRASER, G. On parameter tuning in search based software engineering. In *Symposium on Search-Based Software Engineering, SSBSE 2011* (September 2011).
- [7] ARTZI, S., DOLBY, J., TIP, F., AND PISTOIA, M. Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis* (Trento, Italy, 2010), ISSTA '10, ACM, pp. 49–60.
- [8] ARTZI, S., KIM, S., AND ERNST, M. D. ReCrash: Making software failures reproducible by preserving object states. In *ECOOP '08* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 542–565.
- [9] BAAH, G. K., PODGURSKI, A., AND HARROLD, M. J. The probabilistic program dependence graph and its application to fault diagnosis. In *ISSTA* (2008), pp. 189–200.
- [10] BARRETT, C., AND TINELLI, C. Cvc: A cooperating validity checker. In *Proceedings of the 14th International Conference on Computer Aided Verification* (2002), pp. 500–504.

- [11] BAUDRY, B., FLEUREY, F., AND TRAON, Y. L. Improving test suites for efficient fault localization. In *28th International Conference on Software Engineering* (Shanghai, China, May 2006), ICSE '06, pp. 82–91.
- [12] BERGAN, T., ANDERSON, O., DEVIETTI, J., CEZE, L., AND GROSSMAN, D. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS* (2010).
- [13] BROOKS, R. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18, 6 (1983), 543 – 554.
- [14] BURGER, M., AND ZELLER, A. Minimizing reproduction of software failures. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, July 2011), ISSTA '11, ACM, pp. 221–231.
- [15] CADAR, C., DUNBAR, D., AND ENGLER, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Design and Implementation* (2008).
- [16] CHECKSTYLE. <http://checkstyle.sourceforge.net>.
- [17] CHILIMBI, T. M., LIBLIT, B., MEHRA, K. K., NORI, A. V., AND VASWANI, K. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE* (2009), pp. 34–44.
- [18] CLAUSE, J., AND ORSO, A. A technique for enabling and supporting debugging of field failures. In *ICSE '07* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 261–270.
- [19] CLEVE, H., AND ZELLER, A. Locating causes of program failures. In *ICSE* (2005), pp. 342–351.
- [20] CODEC-22. <https://issues.apache.org/jira/browse/CODEC-22>.
- [21] CODEC-98. <https://issues.apache.org/jira/browse/CODEC-98>.
- [22] DALLMEIER, V., LINDIG, C., AND ZELLER, A. Lightweight defect localization for java. In *ECOOP* (2005), pp. 528–550.
- [23] DALLMEIER, V., AND ZIMMERMANN, T. Automatic extraction of bug localization benchmarks from history. Tech. rep., Universitaet des Saarlandes, Saarbruecken, Germany, June 2007.
- [24] DALLMEIER, V., AND ZIMMERMANN, T. ibugs, 06 2011.
- [25] DIJKSTRA, E. W. *A Discipline of Programming*. Prentice Hall, Inc., 1976.
- [26] DO, H., ELBAUM, S., AND ROTHERMEL, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.* 10 (October 2005), 405–435.

- [27] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M., AND CHEN, P. M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Symp. on Operating Systems Design and Implementation* (2002).
- [28] ECLIPSE. <http://www.eclipse.org>.
- [29] ERNST, M. D. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, 2000.
- [30] FINDBUGS. <http://findbugs.sourceforge.net>.
- [31] FRANCELE, M. A., AND RUGABER, S. The relationship of slicing and debugging to program understanding. In *IWPC* (1999), pp. 106–113.
- [32] FRASER, G., AND ARCURI, A. Evolutionary generation of whole test suites. In *International Conference On Quality Software (QSIC)* (2011), pp. 31–40.
- [33] FRASER, G., AND ARCURI, A. The seed is strong: Seeding strategies in search-based software testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST 2012)* (2012), pp. 121–130.
- [34] FRASER, G., AND ARCURI, A. Whole test suite generation. *IEEE Transactions on Software Engineering* (2012). To appear.
- [35] FRASER, G., AND ZELLER, A. Generating parameterized unit tests. In *ISSTA* (2011), pp. 364–374.
- [36] GANESH, V., AND DILL, D. L. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)* (Berlin, Germany, July 2007), Springer-Verlag.
- [37] GITHUB. <https://github.com/>.
- [38] GODEFROID, P., KLARLUND, N., AND SEN, K. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2005)* (June 2005), pp. 213–223.
- [39] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. A. Automated white-box fuzz testing. In *Proceedings of Network and Distributed Systems Security (NDSS 2008)* (July 2008), pp. 151–166.
- [40] GOOGLECODE. <http://code.google.com>.
- [41] GRADLE. <http://www.gradle.org>.
- [42] GROCE, A. Error explanation with distance metrics. In *10th International Conference of Tools and Algorithms for the Construction and Analysis of Systems* (Barcelona, Spain, 2004), TACAS '04, pp. 108–122.

- [43] GROSS, F., FRASER, G., AND ZELLER, A. Search-based system testing: High coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2012), ISSTA 2012, ACM, pp. 67–77.
- [44] HAO, D., PAN, Y., ZHANG, L., ZHAO, W., MEI, H., AND SUN, J. A similarity-aware approach to testing based fault localization. In *Proceedings of the Conference on Automated Software Engineering* (November 2005), pp. 291–294.
- [45] HAO, D., ZHANG, L., XIE, T., MEI, H., AND SUN, J. Interactive fault localization using test information. *J. Comput. Sci. Technol.* 24, 5 (2009), 962–974.
- [46] HARROLD, M. J., AND ROTHERMEL, G. Aristotle analysis system – siemens programs, hr variants, 06 2011.
- [47] HARROLD, M. J., ROTHERMEL, G., WU, R., AND YI, L. An empirical investigation of program spectra. In *PASTE* (1998), pp. 83–90.
- [48] HAVELUND, K., AND PRESSBURGER, T. Java pathfinder, a translator from java to promela. In *Theoretical and Practical of SPIN Model-Checking* (1999).
- [49] HSU, H.-Y., JONES, J. A., AND ORSO, A. Rapid: Identifying bug signatures to support debugging activities. In *ASE* (2008), pp. 439–442.
- [50] HUTCHINS, M., FOSTER, H., GORADIA, T., AND OSTRAND, T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94* (Los Alamitos, CA, USA, 1994), IEEE Computer Society Press, pp. 191–200.
- [51] IDEA, I. <http://www.jetbrains.com/idea>.
- [52] INKUMSAH, K., AND XIE, T. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE'08: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering* (2008), pp. 297–306.
- [53] JAVAAGENT. <http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html>.
- [54] JAVADIAGNOSTICMXBEAN. <http://docs.oracle.com/javase/6/docs/jre/api/management/extension/com/sun/management/HotSpotDiagnosticMXBean.html>.
- [55] JAVAMXBEANS. <http://docs.oracle.com/javase/tutorial/jmx/mbeans/mxbeans.html>.
- [56] JAVAOPTIONS. docs.oracle.com/javase/6/docs/technotes/tools/windows/java.html#nonstandard.
- [57] JCREATOR. <http://www.jcreator.com>.

- [58] JDEVELOPER. <http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html>.
- [59] JIANG, L., AND SU, Z. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *ASE* (2007), pp. 184–193.
- [60] JIN, W., AND ORSO, A. BugRedux: Reproducing field failures for in-house debugging. In *Proceedings of the 2012 International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE 2012, IEEE Press, pp. 474–484.
- [61] JODATIME-1788282. http://sourceforge.net/tracker/?func=detail&aid=1788282&group_id=97367&atid=617889.
- [62] JODATIME-2487417. http://sourceforge.net/tracker/?func=detail&aid=2487417&group_id=97367&atid=617889.
- [63] JODATIME-2889499. http://sourceforge.net/tracker/?func=detail&aid=2889499&group_id=97367&atid=617889.
- [64] JONES, J., HARROLD, M. J., AND STASKO, J. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering* (Orlando, Florida, May 2002), pp. 467–477.
- [65] JR., F. P. B. *The Mythical Man-Month*. Addison-Wesley, 1982.
- [66] JUNIT. <http://junit.org>.
- [67] KING, J. C. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.
- [68] KO, A. J., DELINE, R., AND VENOLIA, G. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 344–353.
- [69] KO, A. J., AND MYERS, B. A. Finding causes of program output with the Java Whyline. In *CHI* (2009), pp. 1569–1578.
- [70] KOREL, B. Automated software test data generation. *IEEE Transactions on Software Engineering* (1990), 870–879.
- [71] KUSUMOTO, S., NISHIMATSU, A., NISHIE, K., AND INOUE, K. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering* 7, 1 (2002), 49–76.
- [72] LATOZA, T. D., GARLAN, D., HERBSLEB, J. D., AND MYERS, B. A. Program comprehension as fact finding. In *ESEC/SIGSOFT FSE* (2007), pp. 361–370.
- [73] LEE, D., WESTER, B., VEERARAGHAVAN, K., NARAYANASAMY, S., CHEN, P. M., AND FLINN, J. Online multiprocessor replay via speculation and external determinism. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2010).

- [74] LEVENSHTAIN, V. I. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (1966), 707.
- [75] LIBLIT, B., AIKEN, A., ZHENG, A. X., AND JORDAN, M. I. Bug isolation via remote program sampling. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2003)* (June 2003), pp. 141–154.
- [76] LIBLIT, B., NAIK, M., ZHENG, A. X., AIKEN, A., AND JORDAN, M. I. Scalable statistical bug isolation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2005)* (June 2005), pp. 15–26.
- [77] LIU, C., AND HAN, J. Failure proximity: A fault localization-based approach. In *Proceedings of the International Symposium on the Foundations of Software Engineering* (November 2006), pp. 286–295.
- [78] LIU, C., YAN, X., FEI, L., HAN, J., AND MIDKIFF, S. P. SOBER: Statistical model-based bug localization. In *Proceedings of 10th European Software Engineering Conference and 13th Foundations on Software Engineering* (September 2005), pp. 286–295.
- [79] LU, S., LI, Z., QIN, F., TAN, L., ZHOU, P., AND ZHOU, Y. Bugbench: Benchmarks for evaluating bug detection tools. In *SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools* (2005).
- [80] LYLE, J. R., LAAMANEN, M. T., AND CARLSON, N. M. Pest: Programs to evaluate software testing tools and techniques, 06 2011.
- [81] MAKE. <http://www.gnu.org/software/make>.
- [82] MALBURG, J., AND FRASER., G. Combining search-based and constraint-based testing. In *ASE* (2011).
- [83] MATH-367. <https://issues.apache.org/jira/browse/MATH-367>.
- [84] MAVEN. <http://maven.apache.org>.
- [85] MCCONNEL, S. *CODE COMPLETE*, 2nd edition ed. Microsoft, Redmond, Washington, 2004.
- [86] MCMINN, P. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156.
- [87] MEFTAH, B. Benchmarking bug detection tools. In *SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools* (2005).
- [88] MERCURIAL. <http://mercurial.selenic.com>.
- [89] MONTESINOS, P., HICKS, M., KING, S. T., AND TORRELLAS, J. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2009).
- [90] MÜLLER, M. *Information retrieval for music and motion*. Springer, 2007.

- [91] NETBEANS. <http://netbeans.org>.
- [92] PACHECO, C., LAHIRI, S. K., ERNST, M. D., AND BALL, T. Feedback-directed random test generation. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2007), pp. 75–84.
- [93] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Symp. on Operating Systems Principles* (2009).
- [94] PARNIN, C., AND ORSO, A. Are Automated Debugging Techniques Actually Helping Programmers? In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2011)* (Toronto, Canada, July 2011), pp. 199–209.
- [95] PMD. <http://pmd.sourceforge.net>.
- [96] PYTLIK, B., RENIERIS, M., KRISHNAMURTHI, S., AND REISS, S. P. Automated fault localization using potential invariants. In *AADEBUG* (2003), pp. 273–276.
- [97] QI, D., ROYCHOUDHURY, A., LIANG, Z., AND VASWANI, K. Darwin: an approach for debugging evolving programs. In *ESEC/SIGSOFT FSE* (2009), pp. 33–42.
- [98] REITER, R. A theory of diagnosis from first principles. *Artificial Intelligence* 32, 1 (April 1987), 57–95.
- [99] RENIERIS, M., AND REISS, S. P. Fault localization with nearest neighbor queries. In *ASE* (2003), pp. 30–39.
- [100] RÖSSLER, J. Understanding failures through facts. In *PhD Symposium at the 8th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Szeged, Hungary, September 2011), PhD Symposium at ESEC/FSE '11.
- [101] RÖSSLER, J. How helpful are automated debugging tools? In *USER '12: Proceedings of the first Workshop on User evaluation for Software Engineering Researchers* (June 2012).
- [102] RÖSSLER, J., FRASER, G., ZELLER, A., AND ORSO, A. Isolating failure causes through test case generation. In *International Symposium on Software Testing and Analysis* (Jul 2012).
- [103] RÖSSLER, J., ORSO, A., AND ZELLER, A. When does my program fail? In *Proceedings of the 3rd Workshop on Constraints in Software Testing, Verification, and Analysis* (Berlin, Germany, March 2011), CSTVA '11.
- [104] RÖSSLER, J., ZELLER, A., FRASER, G., ZAMFIR, C., AND CANDEA, G. Reconstructing core dumps. In *ICST '13: Proceedings of the Sixth IEEE International Conference on Software Testing, Verification and Validation* (Mar. 2013).

- [105] RUSSELL, S. J., AND NORVIG, P. *Artificial Intelligence: A Modern Approach*. Pearson Education, Inc., 2003.
- [106] SANTELICES, R. A., JONES, J. A., YU, Y., AND HARROLD, M. J. Lightweight fault-localization using multiple coverage types. In *ICSE* (2009), pp. 56–66.
- [107] SHERWOOD, K. D., AND MURPHY, G. C. Reducing code navigation effort with differential code coverage. Tech. rep., University of British Columbia, 2008.
- [108] SOURCEFORGE. <http://sourceforge.net>.
- [109] SPACCO, J., HOVEMEYER, D., AND PUGH, W. Bug specimens are important. In *SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools* (2005).
- [110] TILLMANN, N., AND DE HALLEUX, N. J. Pex — white box test generation for .NET. In *International Conference on Tests And Proofs (TAP)* (2008), pp. 134–253.
- [111] VALMARI, A. The state explosion problem. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models 1491* (1998), 429–528.
- [112] VON MAYRHAUSER, A., AND VANS, A. M. Program understanding - a survey. Tech. rep., Colorado State University, Department of Computer Science, August 1994.
- [113] WEGENER, J., BARESEL, A., AND STHAMER, H. Evolutionary test environment for automatic structural testing. *Information and Software Technology* 43, 14 (2001), 841–854.
- [114] WEISER, M. Programmers use slices when debugging. *Commun. ACM* 25, 7 (1982), 446–452.
- [115] WEYUKER, E. J. On testing non-testable programs. *Computer* 25, 4 (1982), 5–10.
- [116] WICKSTRÖM, G., AND BENDIX, T. The "hawthorne effect"—what did the original hawthorne studies actually show? *Scandinavian Journal of Work, Environment & Health* 26, 4 (2000), 363–367.
- [117] WIKIPROJECTHOSTINGSITES. Comparison of open source software hosting facilities, Feb. 2012.
- [118] WONG, W. E., DEBROY, V., AND CHOI, B. A family of code coverage-based heuristics for effective fault localization. *The Journal of Systems and Software* 83 (2010), 188–208.
- [119] XIE, T., TILLMANN, N., DE HALLEUX, J., AND SCHULTE, W. Fitness-guided path exploration in dynamic symbolic execution. In *International Conference on Dependable Systems and Networks (DSN)* (2009).

- [120] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. SherLog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems* (March 2010).
- [121] YUAN, D., PARK, S., HUANG, P., LIU, Y., LEE, M. M., ZHOU, Y., AND SAVAGE, S. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (2012), OSDI'12, USENIX Association.
- [122] YUAN, D., ZHENG, J., PARK, S., ZHOU, Y., AND SAVAGE, S. Improving software diagnosability via log enhancement. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems* (March 2011).
- [123] ZAMFIR, C., AND CANDEA, G. Execution synthesis: A technique for automated debugging. In *ACM SIGOPS/EuroSys European Conf. on Computer Systems* (2010).
- [124] ZELLER, A. Yesterday, my program worked. today, it does not. why? In *Proceedings of the ESEC/FSE'99, 7th European Software Engineering Conference* (September 1999), vol. 1687 of *Lecture Notes in Computer Science*, Springer, pp. 253–267.
- [125] ZELLER, A. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering* (New York, NY, USA, November 2002), ACM Press, pp. 1–10.
- [126] ZELLER, A. *Why Programs Fail*. Elsevier, 2009.
- [127] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.* 28 (February 2002), 183–200.
- [128] ZHANG, S., ZHANG, C., AND ERNST, M. D. Automated documentation inference to explain failed tests. In *ASE* (2011), pp. 63–72.
- [129] ZHANG, X., GUPTA, N., AND GUPTA, R. Locating faults through automated predicate switching. In *28th International Conference on Software Engineering* (May 2006), ICSE '06, pp. 272–281.