Isolating Failure Causes through Test Case Generation

Jeremias Rößler · Gordon Fraser · Andreas Zeller
Saarland University
Saarbrücken, Germany
{roessler, fraser, zeller}@cs.uni-saarland.de

Alessandro Orso
Georgia Institute of Technology
Atlanta, USA
orso@cc.gatech.edu

ABSTRACT

Manual debugging is driven by experiments—test runs that narrow down failure causes by systematically confirming or excluding individual factors. The BUGEX approach leverages *test case generation* to systematically isolate such causes from a single failing test run—causes such as properties of execution states or branches taken that correlate with the failure. Identifying these causes allows for deriving conclusions as: "The failure occurs whenever the daylight savings time starts at midnight local time." In our evaluation, a prototype of BUGEX precisely pinpointed important failure explaining facts for six out of seven real-life bugs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Diagnostics, Testing tools*

General Terms

Experimentation

Keywords

Automated debugging, test case generation, failure classification, statistical debugging

1. INTRODUCTION

In the past decade, the field of *automated debugging* has made tremendous advances. Most publications in the field focus on *statistical* approaches to defect localization [24, 26, 27, 29, 18, 28, 7]. 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. 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 [34].

The second line of research in automated debugging is formed by *experimental* approaches—techniques that systematically alter applied changes [38], input [41], or object interaction [11] in order to narrow down failure causes to a small fraction of the search space. The most recent work in the field [11], for instance, minimizes a failing execution to a test case encompassing a mere 0.2% of the source code. While such experimental techniques can precisely pinpoint failure causes, they can also alter program behavior in a way that is impossible to achieve in the original setting. Applying delta debugging on program states [39], for instance, easily produces states that are infeasible, and relies on the run-time system (or the programmer's intelligence) to detect inconsistencies.

In this paper, we present a novel approach to automated debugging that combines the mutual benefits of experimental and statistical approaches, yet avoids their respective shortcomings. Our BUGEX prototype

1. provides a *generic automated debugging scheme* that can be parameterized with arbitrary runtime facts (now: branches and values; future: 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 delta debugging.

Our BUGEX results, as detailed in this paper, show that it can pinpoint central facts with *outstanding precision and quality.* As an example, consider the code in Figure 1. This test uses the JODATIME date and time library to convert a local date (October 18, 2009) into a time interval in the Brazilian time zone. This test makes the program fail, but if we change the date or the time zone, everything works fine. What is so special about October 18, 2009 in Brazil?

BUGEX needs nothing more than this test case to start working; indeed, it would ideally be set up to start automatically after a failure, for instance on a build server. Given this test, after a few minutes BUGEX would report a single branch in the JODATIME code—a branch that is taken *only by failing runs* and never by passing runs (Figure 2). The comment preceding such branch explains

```
public void testBrazil() {
  LocalDate date =
    new LocalDate(2009, 10, 18);
  DateTimeZone dtz =
    DateTimeZone.forID("America/Sao_Paulo");
  Interval interval = date.toInterval(dtz);
}
```

Figure 1: JODATIME bug 2487417: `toInterval()` fails when processing October 18, 2009, Brazil time.

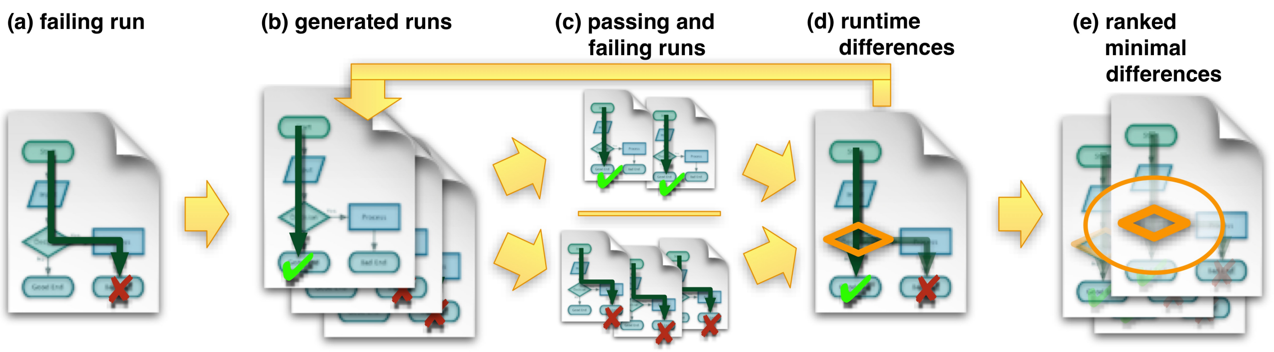

Figure 3: BUGEX in a nutshell. Starting with a failing run (a), BUGEX generates additional runs (b). BUGEX establishes the differences between passing and failing runs (c) in terms of runtime facts (d)—right now, branches taken and state predicates. It statistically ranks these differences and generates more runs (b) to focus on and further minimize these differences. Eventually, BUGEX produces a ranked list of minimal fact differences (e), where the top-ranked facts reveal failure causes.

```
// if the offsets differ,
// we must be near a DST boundary
if (offsetLocal != offsetAdjusted) {
  // ...
}
```

Figure 2: `org.joda.time.DateTimeZone`, lines 863–864.

the cause of the failure: 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. In principle, such a diagnostic quality could also be achieved by a statistical approach, *if* we had a sufficient number of appropriate executions to analyze. Systematically generating these executions is the first advantage of BUGEX over purely statistical approaches.

The basic steps of BUGEX are illustrated by Figure 3. BUGEX starts with a single failure and generates additional runs that are similar to the failing one and either fail or pass. Any difference between these runs in terms of the facts observed during such runs (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 that are strongly correlated with the failure. It then systematically generates more runs that can either further confirm or refute the relevance of a fact. BUGEX is a general approach that could work with any fact that can (1) be either true or false at runtime and (2) be observed. In our current implementation of BUGEX, we considered two kinds of facts: branches covered and properties of the program state. In our experiments we show that BUGEX can pinpoint the failure cause in most cases. In particular, when focusing on branches, BUGEX can pinpoint the failure cause for six out of the seven bugs targeted.

The remainder of this paper is organized as follows. Section 2 starts with an overview on the state of the art. Section 3 describes the BUGEX approach in detail. Section 4 discusses the implementation, in particular the isolation of failure-related branches and state predicates. Section 5 evaluates the approach on seven real-life bugs, both quantitatively and qualitatively. Section 6 closes with conclusion and future work.

2. BACKGROUND

Over the years, researchers have defined increasingly sophisticated debugging techniques, so the body of related work is broad. For the sake of space, here we focus on representative techniques and on the work that is most related to BUGEX.

2.1 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 [24]. Liblit and colleagues extended Tarantula by defining a more general approach that focuses on predicate values instead of statements [26, 27]. Since then, a large number of statistical debugging approaches have been presented in the literature (e.g. [29, 18, 28, 7]). 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.

There are two main issues with this general class of techniques. The first issue 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 [37], 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 test 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.

The second issue with statistical debugging techniques is that 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 [34]. To the best of our knowledge, there are only two statistical debugging techniques that try to address this issue: Rapid, by Hsu and colleagues [20], and Context-Aware Statistical Debugging, by Jang and Su [23]. The former uses a string matching algorithm to identify common segments in the traces of failing executions, whereas the latter combines statistical debugging, clustering, and control-flow analysis to identify relevant control flow paths that may contain bug locations.

Although both techniques mitigate the issue of lack of explanation and context for the statements reported as suspicious, both of them still require large (and adequate) test suites to work.

BUGEX leverages the strength of statistical debugging, while addressing both of its shortcomings. First, it 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. Second, BUGEX investigates many more runtime facts than just statements executed, namely, branches taken (subsuming statement execution) and predicates on the program state. These additional facts can provide a richer explanation of why and how an execution leads to the failure.

2.2 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 [41], code changes [38], or program states [39]. Most related to our work is the concept of *predicate switching,* an approach that flips branch outcomes [43] 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 the null oracle (no oracle at all) in case of errors detected by the runtime system. 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.

2.3 Test Case Generation

Automated test 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; as this is computationally cheap, large numbers of tests can be produced in a short time (e.g., Randoop [33]). As random tests are unlikely to reach parts of the code that require the traversal of complex predicates, more systematic approaches have been presented. In this space, dynamic symbolic execution (DSE) (e.g., [16]) is a prominent solution. DSE collects, 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, one can theoretically explore all possible paths. DSE has been integrated in popular tools, such as Pex [35] and Sage [17]. An alternative approach is to cast the test generation problem as a search problem and use efficient meta-heuristic search algorithms to produce solutions, that is, test cases [31]. 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. For example, our recent EVOSUITE prototype [13] evolves test suites towards satisfying a coverage criterion using a genetic algorithm and is applicable to any Java library that requires no user input. A promising avenue of research in this direction considers combinations of the individual techniques (e.g., [22, 30]).

So far, there has been surprisingly little work leveraging test case generation for debugging purposes (unless one qualifies delta debugging as a test case generator). Baudry and colleagues [10] 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. [9] 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.

2.4 Programmer Support

Most automated debugging techniques discussed so far focus on getting good *quantitative* results in predicting defect locations and are often evaluated using unrealistic benchmarks, such as the Siemens suite [21]. (The Siemens suite is unrealistic due to the small size of its programs, manually crafted one-line bugs, and the excessive size of its test suite, which favors statistical approaches).

Only a few approaches explicitly try to help the developer understand a failure. Tarantula, for instance, visualizes test information and highlights suspicious code [24], but provides no further explanation. The cause-effect chains and cause transitions of Cleve and Zeller [12] explain a failure in terms of how variable values cause each other through a program execution. Zhang et al. [42] 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 for arbitrary runtime facts to correlate with failure, including branches taken (implying the statements executed) or variables taking specific values.

Dialog-oriented approaches also explicitly support failure understanding. Whyline is a tool from Ko and colleagues [25] that allows developers to ask questions about the visual output of a program based on a recorded execution. The tool by Hao and colleagues [19] 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.

3. THE BUGEX APPROACH

We now discuss the BUGEX approach in detail. In the following sections, we discuss the individual steps, as shown in Figure 3; in addition, the pseudocode in Figure 4 presents a schematic view of our approach.

Note that this approach can be implemented with *any* kind of fact that one deems of interest. The only requirements are that (1)

```
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: Simplified pseudocode that depicts our approach.

there is a way to observe whether the facts are true or false for an execution and (2) it must be possible to generate new test cases that evaluate a certain fact differently and reach the failure conditions (without necessarily failing).

3.1 The Failing Run

We start with Step (a) in Figure 3, namely, the input to BUGEX. As detailed in Figure 4, 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. As discussed in Section 2.2, 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.

By default, BUGEX uses *the last branch* before the failure occurs as oracle (Line 1 in Figure 4). All generated test cases must evaluate this last branch b_{last};[1] those that do not are ignored for correlation and discarded. This last branch b_{last} encodes the oracle on the lowest abstraction level; as taking it correlates with failure and success by construction, this branch is excluded from the search.

3.2 Generating Runs

3.2.1 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 our recent EVOSUITE tool, which implements an evolutionary search approach enhanced with dynamic symbolic execution (see Section 2.3). 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 the current version of BUGEX, we extended EVOSUITE to support two kinds of facts: *branches* and *state predicates*.

[1]Technically, 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 paper we use the term branch with both meanings.

Intuitively, the aim of test case generation is to produce tests that (1) are as similar to the failing test case as possible, but (2) differ with respect to specific individual facts. The general underlying motivation for this approach is the notion of *counterfactual causality*, as detailed in [40]. We want an *actual cause* to be a minimal difference: if the cause is present, the failure is present; if the cause is absent, so is the failure. In practice, however, it is difficult to generate test cases that differ in only one fact. As a proxy for that, 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. If the generation is successful, this is a good indicator (depending on what other facts changed), that this fact is irrelevant to the failure. 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.

3.2.2 Test Case Requirements

During operation, BUGEX iteratively chooses a fact (Line 9 in Figure 4) and attempts to derive first a failing test (Line 10) and then a passing one (Line 12) that 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. **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 Sections 4.1 and 4.2. Given such a fitness function, EVOSUITE takes care of deriving test cases that satisfy the above requirements (see [13] for details).

3.2.3 Search Optimizations

To improve the search process, we apply several optimizations:

1. When choosing the next fact to consider for test generation, BUGEX ranks the facts (see Section 3.3) and chooses one of the facts that is highly correlated to the failure (Line 9 in Figure 4). This realizes the feedback loop between test case assessment and test case generation (from (d) to (b) in Figure 3). In addition, in this way we can achieve the quick response times shown later in Section 5 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.
2. BUGEX uses *seeeding* [14], 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) 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 tests with the highest fitness value.

3. 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 that the tests perform.

3.3 Ranking Facts

The retrieval of runtime facts as indicated by the *getFacts* function in Figure 4 (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 [7]. 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 approach showed to produce good results and has the additional benefit that it is computationally cheap.[2]

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.

[2]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 would be split among them, which would result in lower individual rankings (as further discussed in Section 6).

As an example, consider the list of ranked branches for the JODA-TIME Brazilian Date bug introduced in Section 1. The topmost branch at line 864, shown in Figure 2, 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 (many orders of magnitude, actually), BUGEX only reports the first branch in this case.

4. 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*.

4.1 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. Because a passing run by definition does not contain a failure, and 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 [32], a technique originally defined for the alignment of audio tracks. 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* [31] 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 `(arg < 0)` is evaluated with `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.

4.2 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 paper, for simplicity and when not ambiguous, we use the term predicates to refer to state predicates.) The retrieval of these predicates is based on earlier work [15]: 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 (excluding some insensible comparisons, such as constant with constant). The resulting set of predicates represents the concrete predicates that hold for the values at method entry for one particular execution of the method. This can clearly result in a large amount of predicates to consider. For the JODATIME Brazilian bug introduced in Section 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. Note that, although considering only methods in the dynamic slice often tremendously reduces the amount of relevant statements, such statements are scattered throughout the code, so the number of relevant methods is only slightly reduced in some cases. 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. For the JODATIME Brazilian bug introduced in Section 1, for instance, this would mean we would have to investigate the predicates generated for 82 methods. As each predicate for each method represents a fact, 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 or b) are on the stack trace of the failure.

The distance function for an individual fact can again be calculated using a metric similar to the one used for the branch distance. For example, consider the predicate `(x > 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.

4.3 First Branches, then State Predicates

The implementation of BUGEX produces two independent lists: One of failure-related branches, and one of failure-related state predicates; the programmer can choose which one to focus upon first. Branches refer to conditions that not only are domain-specific, but also induce a change in control flow (and thus a likely change in behavior). 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 (Section 5).

5. EVALUATION

5.1 Real-Life Case Study

To assess how well BUGEX works in practice, we implemented it for the two types of facts that we discussed earlier in the paper—branches and state predicates—and conducted a case study on four programs and seven defects. Our selection of programs and defects was driven by the following requirements:

Table 1: BUGEX evaluation subjects

ID	Section	Subject	Lines of code	Given tests
JOD1	5.4.1	Brazilian Date bug	62,326	3,497
VM1	5.4.2	Vending Machine bug	68	1
MAT1	5.4.3	Sparse Iterator bug	53,496	1,580
COD1	5.4.4	Base64 Decoder bug	8,147	1,149
JOD2	5.4.5	Western Hemisphere bug	62,326	3,497
COD2	5.4.6	Base64 Lookup bug	6,154	185
JOD3	5.4.7	Parse French Date bug	53,845	3,392

Table 2: Number of facts reported by BUGEX

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

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 output of BUGEX.

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. (Because BUGEX is not defined in terms of EVOSUITE, these are mainly requirements due to our current implementation.) The second and third ones are fairly standard requirements.

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 1 provides an overview on the program and defect characteristics.

5.2 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.

5.3 Quantitative Results

5.3.1 Numbers of Facts Reported

Let us start with RQ1. The results produced by BUGEX can be found in Table 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 in Section 4.3, 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, the absolute number is not as low as for branches. As we detail in Section 5.4, 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.

5.3.2 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 2 ranges from a handful of seconds to multiple hours.[3]

In practice, however, developers do not have to wait this long for their results. In Figure 5, 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.

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

5.3.3 Statistical Debugging with Supplied Test Suites

[3]All 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.

Table 3: Number of branches reported by statistical debugging

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

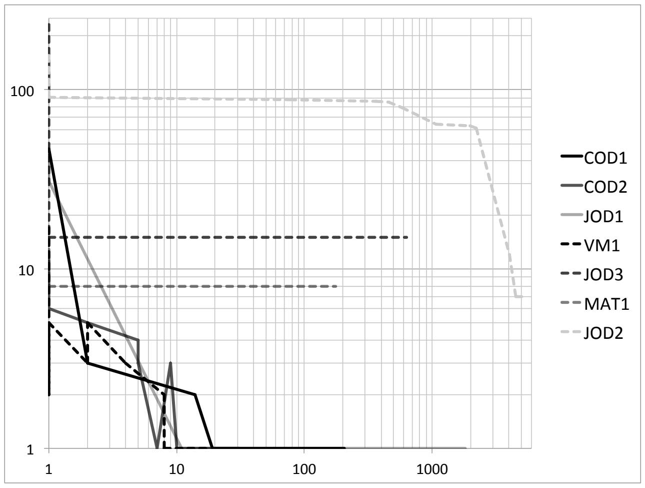

Figure 5: Number of branches remaining over time. The X axis reports time in seconds, the Y axis the number of branches. Note the logarithmic scale on both axes.

To put these numbers into context, we applied the statistical debugging approach on which BUGEX relies (see 3.3) on the branches executed using the *test cases that were supplied with the application.* These results are summarized in Table 3.

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.

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

5.4 Qualitative Results

Let us now discuss RQ2 by assessing whether and to what extent the results help developers in understanding the failure. To do this, we discuss each failure from Table 1, starting with the BUGEX report and relating it to the defect in the code.

5.4.1 JODATIME *Brazilian Date Bug*

This is the failure discussed in Section 1. Bug report 2487417 for JODATIME [1] manifests itself with an exception thrown in the code in Figure 1.

Branches. BUGEX's result (Figure 2) 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 defect in the code to be simply fixed; 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.

5.4.2 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 [11]. 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 problematic if-clause as its sole result:

```
if (this.currValue == 0) {
  this.enabled = false;
}
```

The vending machine fails when the branch 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.

5.4.3 Commons Math Sparse Iterator Bug

Apache Commons Math is a library of lightweight, self-contained mathematics and statistics components. Defect number 367 [2] is revealed by the following test:

```
public void test() {
  double[] vdata = { 0.0, 1.0, 0.0 };
  RealVector vector =
    new ArrayRealVector(vdata);
  Iterator<RealVector.Entry> iter =
    vector.sparseIterator();
  iter.next().getValue();
  iter.next().getValue(); // throws exception
}
```

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. 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 `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.

5.4.4 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 [3] 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`.

Branches. BUGEX reports a single branch as the culprit, namely `buffer != null` in `Base64.readResults()`:

```
int readResults(...) {
  if (buffer != null) {
    // ...
    if (...) {
      // ...
      buffer = null;
    }
    // ...
  }
  return eof ? -1 : 0;
}
```

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 one 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.

5.4.5 JODATIME *Western Hemisphere Bug*

JODATIME bug report 2889499 [4] reads as follows:

`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)`.

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.

5.4.6 *Commons Codec Base64 Lookup Bug*

Issue number 22 in the apache commons codec library [5] 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. 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 `arrayOctect.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.

5.4.7 *JODATIME Parse French Date Bug*

JODATIME's bug report 1788282 says that parsing a valid french date fails with an `IllegalArgumentException` [6].

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* [36], a general issue that affects many software testing activities. From a methodological point of view, it would clearly be highly desirable to have a 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.

5.4.8 *Summary*

After examining the BUGEX results for the seven failures in Table 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 [34], 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.*

5.4.9 *Statistical Debugging*

To have a baseline for our results, we also examined the topmost ranked branches reported by statistical debugging (Table 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 Reference[34] 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. [9]. 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.

5.5 Threats 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 [8]. 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.

6. CONCLUSION AND CONSEQUENCES

When developers debug programs manually, they run tests and experiments to systematically narrow down failure causes. BUGEX is the first approach to automate this process for generic run-time facts. By systematically generating test cases, BUGEX can isolate execution features 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.

Despite these initial successes, the combination of test case generation and automated debugging is still in its infancy. Our 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. 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 fact 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 when `current == null` and `hasNext() == true` hold").

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.

Integration with minimization. JINSI [11] 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. We plan to run an extended user study on BUGEX in the Summer of 2012, which will aim to asses what developers need for efficient debugging, and how well automated debugging tools like BUGEX meet these expectations.

More material on BUGEX is available at

`http://www.st.cs.uni-saarland.de/bugex/`

7. ACKNOWLEDGMENTS.

This research was supported in part by grant Ze509/4-1 from Deutsche Forschungsgemeinschaft and by grants NSF CCF-0916605 and NSF CCF-0964647 to Georgia Tech. Valentin Dallmeier, Nikolai Knopp, Kevin Streit, and Florian Groß provided useful feedback on earlier revisions of this paper.

8. REFERENCES

[1] http://sourceforge.net/tracker/?func=detail&aid=2487417&group_id=97367&atid=617889.
[2] https://issues.apache.org/jira/browse/MATH-367.
[3] https://issues.apache.org/jira/browse/CODEC-98.

[4] `http://sourceforge.net/tracker/?func=detail&aid=2889499&group_id=97367&atid=617889`.

[5] `https://issues.apache.org/jira/browse/CODEC-22`.

[6] `http://sourceforge.net/tracker/?func=detail&aid=1788282&group_id=97367&atid=617889`.

[7] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An observation-based model for fault localization. In *WODA*, pages 64–70, 2008.

[8] A. Arcuri and G. Fraser. On parameter tuning in search based software engineering. In *SSBSE*, pages 33–47, 2011.

[9] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *ISSTA*, pages 49–60, 2010.

[10] B. Baudry, F. Fleurey, and Y. L. Traon. Improving test suites for efficient fault localization. In *ICSE*, pages 82–91, 2006.

[11] M. Burger and A. Zeller. Minimizing reproduction of software failures. In *ISSTA*, ISSTA, pages 221–231, 2011.

[12] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.

[13] G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In *QSIC*, pages 31–40, 2011.

[14] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *ICST*, 2012.

[15] G. Fraser and A. Zeller. Generating parameterized unit tests. In *ISSTA*, pages 364–374, 2011.

[16] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[17] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, pages 151–166, 2008.

[18] D. Hao, Y. Pan, L. Zhang, W. Zhao, H. Mei, and J. Sun. A similarity-aware approach to testing based fault localization. In *ASE*, pages 291–294, 2005.

[19] D. Hao, L. Zhang, T. Xie, H. Mei, and J. Sun. Interactive fault localization using test information. *J. Comput. Sci. Technol.*, 24(5):962–974, 2009.

[20] H.-Y. Hsu, J. A. Jones, and A. Orso. Rapid: Identifying bug signatures to support debugging activities. In *ASE*, pages 439–442, 2008.

[21] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, pages 191–200, 1994.

[22] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE*, pages 297–306, 2008.

[23] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *ASE*, pages 184–193, 2007.

[24] J. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[25] A. J. Ko and B. A. Myers. Finding causes of program output with the Java Whyline. In *CHI*, pages 1569–1578, 2009.

[26] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.

[27] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[28] C. Liu and J. Han. Failure proximity: A fault localization-based approach. In *FSE*, pages 286–295, 2006.

[29] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical model-based bug localization. In *ESEC/FSE*, pages 286–295, 2005.

[30] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *ASE*, 2011.

[31] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[32] M. Müller. *Information retrieval for music and motion*. Springer, 2007.

[33] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84, 2007.

[34] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, pages 199–209, 2011.

[35] N. Tillmann and N. J. de Halleux. Pex — white box test generation for .NET. In *TAP*, pages 134–253, 2008.

[36] E. J. Weyuker. On testing non-testable programs. *Computer*, 25(4):5–10, 1982.

[37] W. E. Wong, V. Debroy, and B. Choi. A family of code coverage-based heuristics for effective fault localization. *The Journal of Systems and Software*, 83:188–208, 2010.

[38] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE*, pages 253–267, 1999.

[39] A. Zeller. Isolating cause-effect chains from computer programs. In *Symposium on Foundations of Software Engineering*, FSE, pages 1–10, 2002.

[40] A. Zeller. *Why Programs Fail*. Elsevier, 2009.

[41] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

[42] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *ASE*, pages 63–72, 2011.

[43] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE*, pages 272–281, 2006.