Automated Documentation Inference to Explain Failed Tests

Sai Zhang[1] Cheng Zhang[2] Michael D. Ernst[1]
[1]University of Washington [2]Shanghai Jiao Tong University
szhang@cs.washington.edu cheng.zhang.stap@sjtu.edu.cn mernst@cs.washington.edu

***Abstract*—A failed test reveals a potential bug in the tested code. Developers need to understand which parts of the test are relevant to the failure before they start bug-fixing.**

This paper presents a fully-automated technique (and its tool implementation, called FailureDoc) to explain a failed test. FailureDoc augments the failed test with explanatory documentation in the form of code comments. The comments indicate changes to the test that would cause it to pass, helping programmers understand why the test fails.

We evaluated FailureDoc on five real-world programs. FailureDoc generated meaningful comments for most of the failed tests. The inferred comments were concise and revealed important debugging clues. We further conducted a user study. The results showed that FailureDoc is useful in bug diagnosis.

I. Introduction

A failed unit test indicates a potential bug in the tested code. A programmer must understand the cause of the failure and confirm its validity before starting bug-fixing. Recently, many automated test generation techniques [2], [20], [21], [26], [28] have been studied to create tests, but few techniques are proposed to *explain* why a test fails. Understanding *why a test fails* or even knowing *which part of the code should be inspected first in debugging* is a non-trivial task. This is a particular problem for automatically-generated tests which are often long and have poor readability, but it is also relevant for human-written tests.

For example, Figures 1 and 2 show a human-written test and an automatically-generated test, respectively. The first test in Figure 1 is associated with a JDK bug report. It shows that method `Arrays.toArray` is not type-safe. However, when executed, this test throws an `ArrayStoreException` at the last statement, which is not obviously related to any type safety issues. To confirm this reported bug, a programmer must manually connect the available failure symptom (an `ArrayStoreException`) with possible failure causes. The second test, shown in Figure 2, is also not easy to understand. This automatically-generated test involves classes such as `ArrayList`, `TreeSet`, and `Collections` in the JDK, each of which contains hundreds of lines of code. Executing this test also does not give much useful information: the assertion simply fails without dumping any stack trace as debugging clues. Furthermore, the test has already been minimized: if any of the calls is removed (and the code is fixed up so that the test compiles again), then the bug is not triggered, and the test passes.

```
public void test1() {
  ArrayList<Number> nums = new ArrayList<Number>();
  Integer i = new Integer(1);
  boolean b0 = nums.add(i);
  Long l = new Long(-1);
  boolean b1 = nums.add(l);
  Integer[] is = new Integer[0];
  //This statement throws ArrayStoreException
  Integer[] ints = nums.toArray(is);
}
```

Fig. 1. A human-written failed test. This test reveals a potential error in the JDK (bug id: 7023484).

```
public void test2() {
  int i = 1;
  ArrayList l = new ArrayList(i);
  Object o = new Object();
  boolean b0 = l.add(o);
  TreeSet t = new TreeSet(l);
  Set s = Collections.synchronizedSet(t);
  //This assertion (reflexivity of equals) fails
  assertTrue(s.equals(s));
}
```

Fig. 2. An automatically-generated failed test. This test reveals an error in JDK version 1.6. It shows a short sequence of calls leading up to the creation of an object that is not equal to itself.

Good documentation (i.e., code comments) can help programmers quickly understand what source code does, facilitating program comprehension and software maintenance tasks [8]. Unfortunately, a human-written test is often poorly documented, and few automated test generation tools can adequately comment the code they generate. Even more importantly, when a test fails, the most useful documentation is relevant to the defect in the tested code and leads the developer to that defect. Later, when the test reveals a different defect, different documentation would be best. As a result of the lack of contextually-relevant documentation, programmers must guess about what parts of the test and the tested code are relevant.

Proposed Approach. This paper presents a fully-automated approach (and its tool implementation, called FailureDoc) to infer documentation for a failed test. FailureDoc is not an automated fault localization tool [13], [14], [29] that pinpoints the exact buggy code. Instead, it augments a failed test with debugging clues: code comments that provide potentially useful facts about the failure, helping programmers fix the bug quickly. Figures 3 and 4 show the inferred documentation for the failed tests of Figures 1 and 2, respectively.

In Figure 3, the comments above lines 5 and 6 reveal important clues that the test passes if object `l` is changed to

```
1. public void test1() {
2.   ArrayList<Number> nums = new ArrayList<Number>();
3.   Integer i = new Integer(1);
4.   boolean b0 = nums.add(i);
     //Test passes if line is: Integer l = new Integer(0);
5.   Long l = new Long(-1);
     //Test passes if l is not added to nums
6.   boolean b1 = nums.add(l);
7.   Integer[] is = new Integer[0];
8.   //This statement throws ArrayStoreException
9.   Integer[] ints = nums.toArray(is);
10. }
```

Fig. 3. The failing test of Figure 1 with code comments inferred by the FailureDoc tool (highlighted by underline).

```
1. public void test2() {
2.   int i = 1;
3.   ArrayList l = new ArrayList(i);
     //Test passes if o implements Comparable
4.   Object o = new Object();
     //Test passes if o is not added to l
5.   boolean b0 = l.add(o);
6.   TreeSet t = new TreeSet(l);
7.   Set s = Collections.synchronizedSet(t);
8.   //This assertion (reflexivity of equals) fails
9.   assertTrue(s.equals(s));
10. }
```

Fig. 4. The failing test of Figure 2 with code comments inferred by the FailureDoc tool (highlighted by underline).

`Integer` type or if `l` is not added to the `nums` list. These clues guide the programmer to discover that the test failure is because the test erroneously adds two type-incompatible objects `i` and `l` into the list, and later casts both of them to `Integer` type. Such information is much more helpful in understanding why a test fails, than merely dumping an `ArrayStoreException`. In Figure 4, the comment above line 4 discloses a crucial fact that the test passes if `Object o` implements `Comparable`. This clue guides the programmer to inspect places where object `o` is used. In fact, the `TreeSet` constructor is buggy: it should not accept a list containing a non-`comparable` object, but it does. This indicates the exact cause and a possible bug fix.

To infer useful documentation, FailureDoc *simulates* programmers' debugging activity. Its design is based on the following common debugging practice: given a failed test, a programmer often tries to make some (minimal) edits to make it pass, observes the difference between passing and failing executions, then generalizes those failure-correcting edits to understand the failure cause. FailureDoc automates the above reasoning process, summarizing its observations as documentation.

FailureDoc works in four phases (Figure 5), namely *value replacement*, *execution observation*, *failure correlation*, and *documentation generation*. In the first phase, FailureDoc first generates an object pool, and then mimics programmers' activity in correcting a failed test by repeatedly replacing existing values with possible alternatives. Each replacement creates a slightly mutated test. In the second phase, FailureDoc executes the mutated test, uses static slicing to prune irrelevant statements, and *selectively* observes its outcomes. In the third phase, FailureDoc uses a statistical algorithm to correlate the replaced values with their corresponding outcomes, identifying suspicious statements and their failure-correcting objects. In the final phase, for each identified suspicious statement, FailureDoc uses a Daikon-like technique [10] to summarize properties of the observed failure-correcting objects, translating them into explanatory code comments.

Evaluation. We implemented the FailureDoc prototype, and evaluated its effectiveness through an experiment and a user study. In our experiment, we used FailureDoc to generate comments for failed tests from five real-world programs. FailureDoc successfully generated human-readable comments for 10 out of 12 failed tests. We validated the helpfulness of the inferred comments by sending the documented tests to the developers. The developers' reaction strongly suggests that FailureDoc is useful in practice. We also conducted a user study to investigate the usefulness of the inferred documentation. With the generated comments, programmers spent 14% less time in understanding the revealed bugs than without those comments; and programmers spent 8.5% less time than with the aid of delta debugging [29].

Contributions. The main contributions of this paper are:

- **Technique.** A technique to infer descriptive comments to explain failed tests (Section II).
- **Tool.** An open-source automated tool implementing the proposed technique (http://code.google.com/p/failuredoc/).
- **Evaluation.** An experiment and a user study demonstrate the usefulness of the proposed technique (Section III).

II. Technique

Figure 5 gives an overview of FailureDoc's architecture. FailureDoc consists of four major modules, working in a pipelined manner.

(1) Value Replacement. This module takes a failed test as input. It first uses a randomized algorithm to create an object pool containing instances of all needed classes, then mutates the failed test by repeatedly replacing expressions in the test code with possible alternatives from the created object pool to construct a set of slightly mutated tests (Section II-A).

(2) Execution Observation. This module executes the mutated tests to obtain execution traces. For each mutated test, this module uses static slicing to prune all irrelevant statements from the execution trace (Section II-B).

(3) Failure Correlation. This module takes as inputs the replaced values and the observed outcomes. It uses a statistical algorithm to identify a small set of suspicious statements (that have a strong correlation with the test failure) and their *failure-correcting* objects (Section II-C).

(4) Documentation Generation. This module generalizes properties of the observed failure-correcting objects for each suspicious statement, then converts the generalized properties into documentation (Section II-D).

A. Value Replacement

Given a failed test, the value replacement module repeatedly replaces expressions in the test code with possible alternatives to construct a set of mutated tests. For example, in Figure 2, FailureDoc can replace `int i = 1` on line 2 with `int i =`

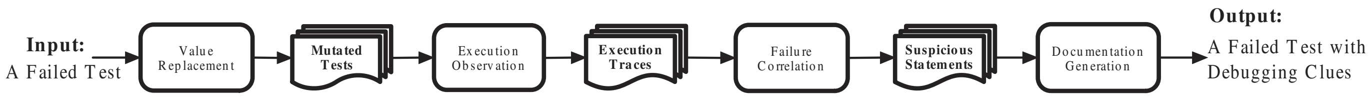

Fig. 5. The architecture of FailureDoc. It augments a failed test with explanatory documentation (debugging clues) in four steps.

0, or replace `TreeSet t = new TreeSet(1)` on line 6 with `TreeSet t = new TreeSet(); t.put(10);`.

To correctly and efficiently implement value replacement, two key challenges must be addressed. First, FailureDoc must create new values to replace an existing one. Second, for the sake of efficiency, FailureDoc must select among the new values it creates for each existing value.

1) Value Generation: The FailureDoc technique can be instantiated using any value generation technique, such as exhaustive generation up to a given size bound [2]. Our current implementation uses an existing random test generation algorithm, that of Randoop [21]. We now briefly describe it.

FailureDoc parses the failed test to extract all referred-to classes. For example, the failed test in Figure 2 refers to classes `Integer`, `Object`, `ArrayList`, `TreeSet`, `Set`, and `Collections`. Optionally, the user can also provide FailureDoc additional classes for value generation, but our experiments did not use this capability.

FailureDoc next uses an existing random test generation algorithm [20], [21] to create object instances of the referred-to classes, and keeps all created objects in a value pool. The algorithm iteratively builds a method-call sequence that produces an object value, by randomly selecting a method or constructor to invoke, using previously computed values as inputs. FailureDoc uses object values in the pool to mutate the failed test. The mutated tests are never shown to the user.

2) Value Selection: After generating an object pool, a natural question is how to select possible alternatives for each expression. Many of the objects in the pool may be similar. Hence, the naive approach of choosing every type-compatible object from the pool to replace the existing value is unnecessary and inefficient. On the other hand, understanding the failure cause requires selecting a group of diverse objects as replacement candidates, in order to expose different test behaviors. Randomly selecting a group of objects from the pool may end up with the same test behavior, which would not be helpful to infer useful information.

To alleviate this problem, FailureDoc *adaptively* selects a diverse set of objects as replacement candidates based on an abstract object profile representation (described below). Adaptive selection reduces the likelihood of choosing a group of similar objects, in which most of the objects reveal the same test behavior while failing to reveal other behaviors.

For each expression e of type T_e in the test code, FailureDoc creates approximately k mutants (k is user-settable; our experiments used the default value k = 20) by replacing e by another value from the created pool. If the pool has k or fewer elements of type T_e, FailureDoc uses them all to create mutants. Otherwise, FailureDoc tries to choose k values that are as different from one another as possible, in the following way.

For each object of type T_e in the pool, FailureDoc computes its abstract object profile. Let a be the number of distinct abstract object profiles. FailureDoc randomly chooses $\lceil k/a \rceil$ values with each abstract object profile.

An *abstract object profile* is a boolean vector that abstracts the object's concrete state (the values of its fields). Each field of the object maps to a distinct set of boolean values in the vector.

- A concrete numerical value v (of type `int`, `float`, etc.), maps to three abstract values $v < 0$, $v = 0$, and $v > 0$.
- A concrete boolean field value v maps to two abstract values $v = \texttt{true}$ and $v = \texttt{false}$.
- A concrete enumeration value v chosen from choices $e_1, \ldots, e_j$ maps to j abstract values, $v = e_i$ for each $1 \leq i \leq j$.
- A concrete object reference value v is mapped to two abstract values $v = \texttt{null}$ and $v \neq \texttt{null}$.
- A concrete array or collection value v is mapped to three abstract values as follows: the same two abstract values as for objects (since an array or collection is an object); one abstract value for whether v is empty.

For example, suppose the `TreeMap` class has two fields `int size` and `Set entrySet`, which represent the map size and the internal data representation, respectively. The following table summarizes the corresponding abstract object profiles for an empty and a non-empty `TreeMap` object. **T** means the property holds.

A `TreeMap` Object	Abstract Object Profile					
	`size`			`entrySet`		
	< 0	$= 0$	> 0	= `null`	$\neq$ `null`	is empty?
an empty map	F	**T**	F	F	**T**	**T**
a non-empty map	F	F	**T**	F	**T**	F

The above abstract object profile, which is used in our implementation, looks at the top level of the concrete representation. The abstract object profile can go as deep in the object as desired. It is straightforward to extend it to more of the concrete representation, as follows:

- Enrich the abstraction of a concrete object reference or enumeration value, by adding abstract values for each of its fields. This corresponds to following two field references: the abstract object profile depends not just on `x.f`, but also on `x.f.g`.
- Enrich the abstraction for arrays/collections, by adding existential abstract values according to the type of its elements. For example, for an array of objects, add one abstract value indicating whether the array contains any

`null` element, and one abstract value indicating whether the array contains any non-`null` element.

B. Execution Observation

FailureDoc next executes the mutated tests, observing their execution outcomes. For each mutated test, FailureDoc records two types of information:

- the test execution outcome
- the value computed by each expression

FailureDoc classifies the test execution outcome into three categories: *pass*, *fail*, and *unexpected exception*. The first two categories represent that a mutated test throws no exception or throws the exact same exception as the original failing test. The *unexpected exception* category represents the scenario that a different uncaught exception is thrown and the test aborts without producing a final result. For example, if FailureDoc replaces `int i = 1` with `int i = -1` at line 2 of Figure 2, an `IllegalArgumentException` will be thrown at line 3 when executing the mutated test, since `ArrayList` requires a non-negative integer as input to its constructor.

When recording expression values in a mutated test, an important problem is which expression values FailureDoc should record. A straightforward approach is to record the runtime values of all expressions in the mutated test. However, the replaced value may *mask* the effects of some previously-created values. So recording them would introduce incorrect noisy data for the follow-up failure correlation phase. For example, if FailureDoc replaces `TreeSet t = new TreeSet(l)` on line 6 of Figure 2 with `TreeSet t = new TreeSet()`, then objects `i`, `l`, and `o` created on line 2–5 will never affect the test execution result. Those object values should not be correlated with the test execution result. Therefore, to correctly observe expression values, such *masking* effects must be identified.

To address this problem, for each mutated test, FailureDoc computes a static backward slice [27] from the assertion statement that fails in the original test to identify expressions whose outcomes may affect the test execution result. Then, FailureDoc only records computed values of those identified expressions. For example, when executing a mutated test created by replacing `TreeSet t = new TreeSet(l)` on line 6 in Figure 2 with `TreeSet t = new TreeSet()`, FailureDoc analyzes the mutated test code and identifies that only the expressions on lines 6 and 7 may affect the assertion on line 9.

FailureDoc records the execution outcomes of a mutated test in a vector $V = \langle v_1, \ldots, v_n \rangle$, where n is the number of statements in the test. If the ith statement is not in the backward slice from the assertion, then $v_i = \mathit{Ignore}$ (*Ignore* means the outcome is not recorded). Otherwise, v_i is the outcome object of the ith statement.

After executing all mutated tests, FailureDoc collects a set of outcome vectors. These will be used as input to the failure correlation module, which identifies suspicious statements that are most likely to cause the failure.

C. Failure Correlation

We devised an offline statistical algorithm that isolates a small set of suspicious statements in a failed test. Our algorithm is a variant of a well-established cooperative bug isolation technique [17]. The basic idea is to identify likely buggy statements by correlating the observed values with the execution results in a set of passing and failing executions.

However, the statistical algorithm described in [17] cannot be directly applied to our problem domain, for two reasons. (1) The original algorithm uses the boolean value of an instrumented predicate as the feature vector to identify likely buggy *predicates*, while FailureDoc needs to use multiple observed values (in an outcome vector) to isolate suspicious *statements*. (2) Merely identifying suspicious statements is not sufficient for FailureDoc. FailureDoc also needs to associate each identified statement with a set of *failure-correcting* objects, using any of which to replace the existing value will make the test pass. The properties of such *failure-correcting* objects will be generalized and translated into human-readable comments by the follow-on documentation generation module.

To address the above two limitations, we first define a new metric *Pass* and re-define three existing metrics $\mathit{Context}$, $\mathit{Increase}$, and $\mathit{Importance}$ as proposed in [17], then present a new statistical algorithm at the end of this section. Interested readers can refer to [17] for more details on the design of the three existing metrics.

For the ith statement s_i in a failed test, there are j different replacing values recorded in outcome vectors, which we denote as v_{i1}, v_{i2}, …, v_{ij}. For statement s_i, we define its *failure-correcting* object set $FC_i = \{v_{ik} \mid$ using v_{ik} to replace the existing value makes the test pass$\}$.

Let $Pr(A|B)$ denote the conditional probability of the event A given event B. Let $S(v_{ij})$ be the number of successful runs in which value v_{ij} replaces the ith statement, and let $F(v_{ij})$ be the number of failing runs in which value v_{ij} replaces the ith statement. The probability of the test passing when using v_{ij} to replace the existing value in the ith statement is:

$$\mathit{Pass}(v_{ij}) = \frac{S(v_{ij})}{S(v_{ij}) + F(v_{ij})}$$

A lower score shows weaker correlation between v_{ij} and the test failure. However, a single metric is often insufficient to identify suspicious code [17], which we also experimentally verified in Section III-A for our problem domain. The major reason is that $\mathit{Pass}(v_{ij})$ does not consider the number of available executions and code execution context. Intuitively, for two replacing values v_{i1} and v_{i2}, if $\mathit{Pass}(v_{i1}) = \mathit{Pass}(v_{i2})$ but v_{i1} is observed in more executions, we should have a stronger belief that v_{i1} is correlated with the failure. Additionally, the observation of v_{ij} could be affected by its execution context. In some cases, high $\mathit{Pass}(v_{ij})$ does not necessarily mean v_{ij} is an interesting *failure-correcting* object, it is possible that the decision that eventually causes the test to pass is made earlier, and the high $\mathit{Pass}(v_{ij})$ score just reflects the fact that this value is observed after the decision has been made.

Input: a failed test t
Output: a set of suspicious statements $Stmts$, each of which is associated with a set of failure-correcting objects

```
1: $Stmts \leftarrow \emptyset$
2: for each statement $s_i$ in $t$ do
3:     $FC_i \leftarrow \{v_{ij} \mid S(v_{ij}) > 0 \wedge F(v_{ij}) = 0$
                $\wedge Increase(v_{ij}) > 0 \wedge Importance(v_{ij}) > k\}$
4:     if $FC_i \neq \emptyset$ then
5:         $Stmts \leftarrow Stmts \cup \langle s_i, FC_i \rangle$
6:     end if
7: end for
8: return $Stmts$
```

Fig. 6. Algorithm for isolating a set of suspicious statements. Each isolated statement is associated with a failure-correcting object set. The threshold k for the $Importance$ metric is user-settable; our experiments use its default value 0.81.

For those reasons, we re-define metrics $Context$, $Increase$, and $Importance$ by considering the code execution context and number of available executions, as follows.

$Context(v_{ij}) = \Pr(\text{Test Passes} \mid \text{the ith line is observed})$

In the above definition, $Context(v_{ij})$ is the probability that the output value of the ith line is recorded (has not been pruned after performing slicing in Section II-B) in a passing execution. We define $Context(v_{ij})$ as:

$$Context(v_{ij}) = \frac{SR(i)}{SR(i) + FR(i)}$$

In the above definition, $SR(i)$ is the number of passing executions when the output of the ith statement is recorded, and $FR(i)$ is the number of failing executions when the output of the ith statement is recorded. Then, we re-define $Increase(v_{ij})$ as:

$$Increase(v_{ij}) = Pass(v_{ij}) - Context(v_{ij})$$

A value v_{ij} with $Increase(v_{ij}) \leq 0$ is unlikely to be a *failure-correcting* object and will be discarded by FailureDoc.

We finally re-define the $Importance$ metric. It considers the number of available passing executions $S(v_{ij})$, to avoid only representing special cases (observed from a small number of executions) but not being applicable to more general cases.

$$Importance(v_{ij}) = \frac{2}{\frac{1}{Increase(v_{ij})} + \frac{1}{\log(S(v_{ij}))}}$$

Figure 6 shows our algorithm for isolating suspicious statements. A statement is classified as suspicious iff its failure-correcting object set $FC \neq \emptyset$.

D. Documentation Generation

For each identified suspicious statement, FailureDoc uses a Daikon-like [10] technique to summarize common properties of its *failure-correcting* objects, and convert the summarized properties into human-readable comments. Each comment indicates a different way to cause the failed test to pass. Thus, even if the comments provide different information, the comments are never in conflict with one another.

1) Object Property Generalization: FailureDoc reports properties that are true over all failure-correcting values. The essential idea is to use a generate-and-check algorithm to test a set of pre-defined potential properties against the values, and then report those properties that are not falsified.

Given a set of objects $FC = \{o_1, o_2, \ldots, o_n\}$, FailureDoc checks the following properties:

- **Type:** do all objects in FC have the same type? For example, are all objects `Comparable`, or of `Collection` type?
- **Abstract Object Profile:** do all objects in FC have the same abstract object profile (Section II-A2)?

The properties can be viewed as forming a lattice based on subsumption (logical implication). The FailureDoc implementation takes advantage of these relationships in order to improve both performance and the intelligibility of the output. For example, if all objects in FC are both integer type and comparable, FailureDoc will suppress the weaker property: being comparable is logically implied by being of integer type.

We produced the list of properties in the abstract object profile by proposing a basic set that seemed natural and generally applicable, based on our debugging experience. We later added other properties we found useful in revealing debugging clues (e.g., checking whether a collection object is empty or not). Compared to the invariant detection engine implemented in Daikon [10], the property set checked by FailureDoc is highly tailored for the debugging purpose. It discards many scalar properties that are extensively used in Daikon, and adds some properties that Daikon lacks, such as checking whether all objects have the same abstract object profile.

Consider the example in Figure 2, and suppose the failure correlation module identifies line 4 as suspicious. FailureDoc will take its failure-correcting object set as inputs, such as { `"hi"`, `100`, `(byte)1`}, and summarize the property that all objects are `Comparable`.

2) Documentation Summarization: FailureDoc concatenates all reported properties and converts them into descriptive documentation. It employs a small number of simple translations to phrase common Java idioms. The translated documentation is presented like a description of *property* as follows:

- `x = null` **becomes** "x is set to: `null`"
- `x == (Integer)0` **becomes** "the line is: `Integer x = new Integer(0);`" (when the type of `x` is not `Integer` in the failed test)
- `x instanceof Comparable` **becomes** "x implements `Comparable`"
- `x.add(o)` returns `false` **becomes** "`o` is not added to `x`" (when `x` is a `Collection` type object)
- `x` has an abstract profile `TreeMap⟨size = 0, entrySet≠null⟩` **becomes** `x` is a `TreeMap` object, in which `size` is `0` and `entrySet` is not `null`

As shown in Figure 4, the translated documentation (code comment) is presented in the form of: "Test passes if *property*". If multiple properties are reported, FailureDoc generates

Program (version)	Program size			Failed tests		Commented tests		FailureDoc execution details	
	LOC	Classes	Methods	Tests	Statements	Tests	Comments	Mutants	Time (seconds)
Time And Money (0.51)	2372	29	492	2	81	1	3	993	114
Apache Commons Primitives (1.0)	9368	210	1739	2	150	2	3	4740	1079
Apache Commons Math (2.2)	14469	131	1333	3	144	2	13	2037	271
Apache Commons Collections (3.2.1)	55400	445	5350	3	83	3	6	1987	780
`java.util` package (1.6.0_12)	48026	191	3387	2	27	2	4	734	29
Total	129635	1006	12301	12	485	10	29	10491	2273

Fig. 7. Subject programs and experimental results in evaluating FailureDoc. Column "LOC" is the number of lines of code, as counted by LOCC [19]. Column "Commented tests" is the number of failed tests for which FailureDoc can infer documentation. Column "Comments" is the total number of inferred comments for the documented tests (one comment per suspicious statement). "Mutants" is the total number of mutated tests created by the value replacement module (Section II-A). Column "Time" is the total time (in seconds) used in the whole documentation inference process, including the time spent on tests for which FailureDoc cannot infer documentation.

documentation in the form of "Test passes if *property 1* or *property 2*".

III. Evaluation

We have implemented a tool, called FailureDoc, and investigated the following two research questions:

1) **RQ1:** Can FailureDoc scale to realistic programs, and generate meaningful documentation for failed tests?
2) **RQ2:** Does the inferred documentation help programmers to understand failed tests?

To answer these research questions, we designed an experiment and a controlled user study.

1) For **RQ1**, we applied FailureDoc to five real-world programs to infer explanatory documentation for failed tests (Section III-A). We examined the inferred documentation and also sent it to the subject developers for feedback.

2) For **RQ2**, we designed a controlled user study to investigate whether the generated documentation aids bug diagnosis (Section III-B). The user study involved 16 participants with an average of 4.1 years of Java programming experience.

A. Experiment: Inferring Debugging Clues

FailureDoc takes as input a failed test. We generated failed tests by running Randoop [21] on five real-world subject programs (Apache Commons Collections[1], Primitives[2], Math[3], Time and Money[4], and `java.util`[5]). Randoop checked 5 default programming rules defined in Java such as the symmetry property of equality: `o.equals(o)`. Randoop outputted 12 failed tests; each one indicates a distinct, real bug in a subject program. On average, each test has 41 lines of code excluding assert statements. Next, FailureDoc inferred documentation for each failed test. Finally, we examined FailureDoc's output to judge the quality of inferred documentation. Figure 7 summarizes the experimental results.

Results. Like the example in Figure 2, most failed tests involve complex code interactions between multiple classes. It was hard for us to tell the failure cause by simply looking at the source code or executing the test.

FailureDoc successfully inferred documentation for 10 of 12 failed tests. The comments FailureDoc creates are reasonably succinct, approximately 1 comment per 17 lines of test code. FailureDoc is fast enough for practical use, taking 189 seconds on average to infer documentation for one failed test.[6] The time used to infer documentation for a test is roughly proportional to the length of the test, rather than the size of the tested program. Most of the time is spent performing value replacement, because reflectively executing a failed test with multiple input values takes a considerable amount of time.

Example. Figure 8 shows a documented test from subject Apache Commons Collections. The comments indicate which part of the test code programmers should inspect first, while ignoring other irrelevant method calls and variables.

In Figure 8, the comments indicate three ways to correct this failed test by either changing the value of `s0` on line 26 from `(Short)1` to `(Integer)0`, or not adding `s0` to `listOrderedSet0` on line 27, or not adding `i5` to `listOrderedSet11` on line 59. In particular, the comment above line 26 reminds programmers the test will pass if an existing value is added to `listOrderedSet0` (the same value as the `new Integer(0)` object defined on line 24, is added to `listOrderedSet0` on line 25). This information guides programmers to check the code inside method `ListOrderedSet.add(Object)`. After further inspection, programmers would find the `add` method does not update a `ListOrderedSet` object state correctly when adding a new element: it only adds the element to the `collection` field, but forgets to update the `setOrder` field. Note that `listOrderedSet9` is just a wrapping object of `listOrderedSet0` on line 23. Thus, updating `listOrderedSet0` also changes the state of `listOrderedSet9`. Thus, this comment indicates the exact cause for the assertion failure.

FailureDoc failed to infer comments for two failed tests, in subjects Time And Money and Apache Commons Math. That is mainly due to the characteristics of the revealed bugs. In Time And Money, the static factory method `everFrom` in class

[1] Apache Commons Collections: http://commons.apache.org/collections/

[2] Apache Commons Primitives: http://commons.apache.org/primitives/

[3] Apache Commons Math: http://commons.apache.org/math/

[4] Time And Money: http://sourceforge.net/projects/timeandmoney/

[5] JDK 1.6: http://download.oracle.com/javase/6/docs/api/

[6] The experiment used a 2.67GHz Intel(R) Core 2 PC with 2GB physical memory (512MB is allocated for the JVM), running Fedora 12.

```
public void testListOrderedSet() throws Throwable {
1.  ListOrderedSet listOrderedSet0 = new ListOrderedSet();
    ...
23. ListOrderedSet listOrderedSet9
      = ListOrderedSet.decorate((Set)listOrderedSet0, list6);
    ...
24. Integer i2 = new Integer(0);
25. boolean b3 = listOrderedSet0.add((Object)i2);
    //Test passes if line is: Integer s0 = new Integer(0);
26. Short s0 = new Short((short)1);
    //Test passes if s0 is not added to listOrderedSet0
27. boolean b4 = listOrderedSet0.add((Object)s0);
    ...
33. ListOrderedSet listOrderedSet11 = new ListOrderedSet();
    ...
58. int i5 = listOrderedSet21.size();
    //Test passes if i5 is not added to listOrderedSet11
59. boolean b8 = listOrderedSet11.add((Object)i5);
    ...
    //this assertion (transitivity of equals) fails
61. assertTrue(listOrderedSet11.equals(listOrderedSet9)
      == listOrderedSet9.equals(listOrderedSet11));
}
```

Fig. 8. A failed test for Apache Commons Collections. FailureDoc automatically augmented it with the underlined debugging comments.

`TimeInterval` incorrectly passes a *hard-coded* `null` value to a constructor. In subject Apache Commons Math, the revealed bug is due to the fact that two objects `new Double(0.0d)` and `new Double(-0.0d)`, though numerically equal, have different hash codes. For both failed tests, there is no way for client code to use value replacement to correct them. Therefore, FailureDoc fails to infer useful code documentation.

Feedback from Developers. We sent the documented failed tests to the subject program developers, asking them to judge the documentation quality. As of August 2011, we received several pieces of positive feedback from the developers. Luc Maisonobe, a developer of Apache Commons Math gave us the following comment:

> *I think these comments are helpful. They give a hint about what to look at. In both (failed test) cases, I had to run the test in a debugger to see exactly what happened but the comment showed me exactly the variable to look at.*

All reported bugs and documented tests are publicly available at: http://www.cs.washington.edu/homes/szhang/failuredoc/bugreports/.

Experimental comparison with delta debugging. Delta debugging [29] is a general technique to isolate failure-inducing inputs. It has the potential to isolate suspicious statements from a failed test, just as our statistical algorithm (Section II-C) does.

We implemented the isolating delta debugging algorithm described in [29], and applied it to the 12 failed tests in Figure 7. Delta debugging [29] experimentally validates whether a certain statement is suspicious, by removing it from the failed test code and re-executing the simplified test, to see whether the same failure occurs again. One limitation of delta debugging is that it cannot isolate statements whose removal would cause a compilation error.

Delta debugging isolated 4 suspicious statements in 3 failed tests (such as, statement 5 in Figure 4, and statements 27 and 59 in Figure 8). By comparison, our statistical algorithm isolated 29 suspicious statements in 10 failed tests, including all 4 statements isolated by delta debugging.

Even more importantly, the statements isolated by delta debugging gave limited information for understanding the failure. For instance, in Figure 4, the comment above statement 5 indicates that the `Object o` defined on line 4 is related to the failure, but does not tell *why the test fails*. In contrast, FailureDoc additionally isolates statement 4 as suspicious, and generates a comment for it, guiding programmers to better understand the failure cause.

Section III-B empirically compares delta debugging with FailureDoc via a user study to show that delta debugging produces less useful results in understanding the failure cause.

Experimental comparison with a single metric statistical algorithm. We next verified the necessity of using multiple metrics ($Pass$, $Increase$, and $Importance$ defined in Section II-C) in isolating suspicious statements. We implemented a simpler statistical algorithm only using metric $Pass$, and compared its results with FailureDoc's output. This simpler algorithm changes line 3 of Figure 6 to

$$FC_i \leftarrow \{v_{ij} \mid Pass(v_{ij}) > 0 \wedge F(v_{ij}) = 0\}$$

This simpler algorithm isolated 32 suspicious statements from the 10 failed tests in Figure 7: the 29 suspicious statements isolated by our algorithm, plus 3 additional statements that are implied by FailureDoc's output. The 3 additional statements are further from the root cause, and are less useful in understanding the test behavior.

B. User Study: Understanding Failed Tests

We performed a controlled user study to investigate two questions. First, can FailureDoc help programmers understand a failed test and fix the revealed bug? Second, is the information provided by FailureDoc more useful than delta debugging [29], a state-of-the-art automated debugging technique?

The participants were given test cases and asked to understand and fix the underlying error. There were three experimental treatments: some test cases were as produced by Randoop; some had been annotated with suspicious statements identified by delta debugging; and some had been annotated with FailureDoc documentation.

Setup. The participants were 16 graduate students majoring in computer science. On average, they had 4.1 years of Java programming experience (min = 1, max = 7, sd = 1.7), and 1.9 years of JUnit experience (min = 0.1, max = 4, sd = 1.5). None of them was familiar with the subject code. Before the user study started, we gave each participant a 15-minute tutorial about the basic concept of JUnit tests, the default contracts that Randoop's JUnit tests check, and (when relevant) the format of FailureDoc's inferred documentation.

Each participant was given 30 minutes per failed test case. For each failed test, we gave the participant two goals. First, participants were asked to *understand* why the test fails, write down its failure cause, and tell us when they had found it. Second, participants were asked to write a patch to *fix* the

Goal	Success Rate		Time Used (in minutes)							
	JUnit	FailureDoc	JUnit				FailureDoc			
			mean	sd	max	min	mean	sd	max	min
Understand Failure	75%	75%	22.6	7.4	30	11	19.9	10.0	30	2
Understand Failure + Fix Defect	35%	35%	27.5	4.8	30	17	26.9	6.9	30	5

(a) Comparison of original un-documented failed tests (column "JUnit") with FailureDoc-documented tests (column "FailureDoc").

Goal	Success Rate		Time Used (in minutes)							
	Delta debugging	FailureDoc	Delta debugging				FailureDoc			
			mean	sd	max	min	mean	sd	max	min
Understand Failure	75%	75%	21.7	8.1	30	6	20.0	9.6	30	2
Understand Failure + Fix Defect	40%	45%	26.1	6.1	30	9	26.5	6.8	30	5

(b) Comparison of tests annotated with faulty statements isolated by Delta debugging [29] (column "Delta debugging") with FailureDoc-documented tests (column "FailureDoc").

Fig. 9. User study results. "Success Rate" represents the percentage of finished tests for a certain goal. "Time Used (in minutes)" represents the average time to complete one task. We used 30 minutes (the maximum allowed time) for participants who failed to complete a certain goal.

defect. After all participants finished their tasks, we checked the correctness of the identified failure cause (the first goal) and the proposed bug fixes (the second goal).

Each participant was assigned to understand and fix 2–4 different failed test cases chosen from Figure 7. 5 participants received two test cases with FailureDoc documentation, then two un-documented test cases. 5 participants received first two un-documented test cases, then two FailureDoc-documented test cases. 3 participants received 1 FailureDoc-documented test case, then 1 test case annotated with suspicious statements identified by delta debugging. 3 participants received 1 delta-debugging-annotated test case, then 1 FailureDoc-documented test case. We assigned fewer participants to the delta debugging treatment, because delta debugging identified suspicious statements for only 3 failed tests. For the other 7 failed tests, we re-used the experimental data for the un-documented test cases. Using this approach, each sub-table of Figure 9 compares the same experimental subjects, which avoids conflating individual differences with treatment differences.

Results. Figure 9 summarizes the user study results.

As shown in Figure 9(a), participants using FailureDoc-documented tests and un-documented tests completed the same number of tasks. However, participants using FailureDoc-documented tests understood the failure cause 14% faster (2.7 minutes faster) than the participants using un-documented tests. For each defect, participants with FailureDoc-inferred documentation spent slightly less time (0.6 minute) to fix, than participants without the documentation. This indicates that the explanatory documentation is most useful for understanding the failure cause.

As shown in Figure 9(b), participants using FailureDoc-documented tests completed more tasks than participants with the aid of delta debugging (40% versus 45% success rate in fixing defects). The two treatments led to the same success rate in understanding a failed test, but participants using FailureDoc-documented tests spent 8.5% less time (1.7 minutes) to do so. The delta debugging treatment led to slightly faster fixes than with FailureDoc (0.4 minute), despite delta debugging's lower success rate. This is primarily due to one programmer who fixed one defect extremely quickly under the delta debugging treatment. In retrospect, setting the maximum permitted time to be larger would have yielded more discriminating results: the average *successful* bug fix (across all treatments) took 23.2 minutes, which is not much less than the 30 minutes that the statistics use for unsuccessful participants.

Participants' feedback. After the user study, we asked all participants to complete a survey, writing down their feedback and suggestions on the inferred documentation. 15 out of 16 participants thought unfamiliarity with the subject code was the major reason for their slow progress or their failure to fix the defect. When asked to classify the usefulness of inferred documentation as either *very useful*, *useful*, *not very useful*, and *not useful at all (misleading)*, 1 participant thought the documentation was *very useful*, 12 out of 16 participants thought the documentation was *useful*, while the remaining 3 participants thought the documentation was *not very useful*.

The 13 participants who found the documentation useful (or very useful) have an average of 4.3 years of Java programming experience, while the remaining 3 participants have an average of 2.6 years of experience. Two participants said they expected the comments to exactly pinpoint the buggy code (that is not the goal of FailureDoc), and thus thought the inferred documentation less useful, and the other participant said the comments were not very useful if he was not familiar with the tested program. More experienced participants could leverage their own experience with the given hint to find the right code to inspect.

During the study, one participant accidentally overlooked a comment in the assigned test, and spent over 25 minutes in understanding one failed test. However, as soon as he noticed the overlooked comment, he understood why the test fails. That participant gave us the following comment: *I should have noticed the comments (earlier). The comment at line 68 did provide information very close to the bug!*

In contrast, another participant efficiently leveraged the inferred documentation, understood the failure cause and proposed a good fix for the failed test in less than 5 minutes. He gave us the following comment: *The comments are useful, because they indicate which variables are suspicious, and help me narrow the search space.* The participant who thought the comment was very useful said: *this kind of comments can help to eliminate the wrong search path and the possibility of missing a bug, thus reducing the debugging time*.

The three participants who thought the comments were not very useful said the information provided by FailureDoc interfered with their established debugging habits. They gave three pieces of negative comments. One participant said: *the comments usually provide information about data and control dependencies, but their meanings are not so obvious for bug fixing. They can be more useful, if they are more descriptive in natural language*. Another participant said that his assigned test was already very simple, and though the comments helped him understand what the test was doing, they were not so useful. The third participant said: *the comments, though [they] give useful information, can easily be misunderstood, when I am not familiar with the [program]*.

After the user study, we asked the 6 participants who received both FailureDoc-documented tests and tests annotated with suspicious faulty statements by delta debugging to judge which information is more useful. All 6 participants thought FailureDoc provided richer and more useful information than delta debugging, since the documentation not only indicated which statements are relevant to the bug, but also gave hints on why the test failed.

Summary. We have three chief findings: (1) compared to an undocumented test, FailureDoc's inferred documentation slightly speeds up the task of understanding and fixing a bug, (2) compared to delta debugging, FailureDoc slightly speeds up the task of understanding a bug, and leads to greater success in fixing the bug, and (3) the inferred documentation is more useful for more experienced programmers.

Threats to validity. There are three major threats to validity. First, the five programs and the diagnosed bugs may not be representative. Thus, we can not claim the results can be extended to an arbitrary program. Second, the generality of our user study is obviously limited: this was a small task, a small sample of people, limited time, and unfamiliar code. Third, the differences in the means are relatively small and not statistically significant, in part because of the 30-minute time limit. These three threats can be reduced by performing experiments on more subjects and users.

IV. Related Work

We next discuss closely-related work on automated software testing, error explanation, statistical program analysis, and automated documentation inference for source code.

Automated software testing. Many automated testing techniques and tools [5], [7], [20], [21], [30], [31] have been developed to find defects in software. Tools like Eclat [20] and Randoop [21] generate random method-call sequences for object-oriented programs. Previous test generation tools have been demonstrated to be promising in finding new bugs, but none of them can explain a failed test. With the wider adoption of automated testing tools, quickly understanding failed tests becomes a demanding requirement. This paper addresses this problem with the FailureDoc tool, to infer descriptive code comments as debugging clues.

ARTOO [5], an adaptive random test generator, uses a distance metric to select a diverse set of objects for test case creation. The metric does not map each concrete field value to an abstract domain. Instead, it directly uses the concrete field values to compute distance between two objects. In contrast, the abstract object profile used in FailureDoc characterizes a Java object at a coarser granularity, by mapping each concrete value to a much smaller abstract domain. This is appropriate because FailureDoc uses the abstract object profile to distinguish two Java objects, instead of computing their distance as ARTOO does.

Program Error Explanation. Explaining a counterexample [1], [12], [16] is fundamentally different from FailureDoc's goal. Representative work like [12] is similar in spirit to delta debugging [29]. It takes an error trace produced by a model checker, and computes a minimal transformation between error and correct traces by conducting a modified binary search over program states. In contrast, FailureDoc takes as input a failed test, creates additional execution traces, and generalizes failure-correcting edits as documentation to explain the test failure. FailureDoc is also different from the state-of-the-art fault localization techniques [6], [13], [14], [29], which can pinpoint the likely buggy code. FailureDoc augments a failed test with debugging clues, helping programmers understand the failure cause and fix the bug.

Statistical Program Analysis. Recently, statistical algorithms have been applied to the program analysis domain [9], [17], [18]. Podgurski et al. [9] applied statistical feature selection, clustering, and multivariate visualization techniques to the task of classifying software failure reports. The CBI project [17] analyzes execution traces collected from deployed software to isolate software failure causes. Specifically, the program is instrumented to collect information from certain run-time values and this information is passed on to a statistical engine to compute likely buggy predicates. Compared to the CBI project, FailureDoc has several differences and a rather different goal. First, instead of having many collected execution traces, FailureDoc is given only one failed test (i.e., a single failing execution trace). It needs to construct extra execution traces before applying a statistical algorithm. FailureDoc addresses this problem by using value replacement to construct slightly mutated tests, and then obtain extra execution traces. Second, the CBI project uses the value (true/false) of an instrumented predicate as the feature vector, while FailureDoc uses a finer granularity: it observes and records multiple computed values of related expressions. Third, FailureDoc does not track the usually long execution trace across the whole program as CBI does. FailureDoc's static analysis and runtime monitoring is intra-procedural, permitting lower overhead. Fourth, CBI and FailureDoc have different goals. CBI uses statistical algorithms to identify likely buggy predicates as final output, while FailureDoc identifies a set of *suspicious* statements and their *failure-correcting* objects for documentation inference.

Documentation Inference for Source Code. There has been some limited work on automated documentation inference for source code. Semi-automated approaches like [22], [23] either determine un-commented code segments and prompt developers to enter comments, or generate comments from user-provided high level abstractions. Some techniques generate comments for exceptions [3], API function cross-references [11], software changes [4], [15], and descriptive summary comments for methods [24], [25]. However, none of the previous work generates documentation to explain a failed test. The difficulties raised in understanding a failed test motivate this work.

V. Conclusion and Future Work

We have presented FailureDoc, a fully-automated technique to infer documentation to explain failed tests. FailureDoc uses a combination of several lightweight techniques to achieve its goal. In our experiment, FailureDoc successfully inferred documentation for 10 out of 12 failed tests from five real-world programs, showing good scalability. The documentation inferred by FailureDoc revealed important information about the test failure, guiding programmers to inspect the right code place. Our user study and developers' reaction further demonstrated its usefulness.

As future work, we plan to investigate alternative strategies in performing value replacement, particularly for failed tests where multiple values need to be replaced to make it pass. We also plan to use more sophisticated techniques like [1] to guide documentation inference, making the tool more accurate and efficient.

Acknowledgements We thank the participants in the FailureDoc user study. We also thank Werner Dietl for performing a careful check on the experimental data. This work was supported in part by ABB Corporation and by NSF grant CCF-0963757. Cheng Zhang's work was supported in part by National Natural Science Foundation of China (NSFC) grants 60673120 and 60970009.

References

[1] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard Trefler. Explaining counterexamples using causality. In *Proc. CAV'09*, pages 94–108, 2009.

[2] C Boyapati, S Khurshid, and D Marinov. Korat: automated testing based on Java predicates. In *ISSTA '02*, 2002.

[3] R P.L. Buse and W R. Weimer. Automatic documentation inference for exceptions. In *ISSTA '08*, pages 273–282, 2008.

[4] Raymond P.L. Buse and Westley R. Weimer. Automatically documenting program changes. In *ASE '10*, 2010.

[5] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. ARTOO: adaptive random testing for object-oriented software. In *ICSE '08*, pages 71–80, 2008.

[6] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *TOSEM*, 17(2):1–37, April 2008.

[7] V Dallmeier, N Knopp, C Mallon, S Hack, and A Zeller. Generating test cases for specification mining. In *ISSTA '10*, 2010.

[8] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *Proc. SIGDOC '05*, pages 68–75, 2005.

[9] W Dickinson, D Leon, and A Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE'01*, pages 339–348, 2001.

[10] M D. Ernst, J Cockrell, W G. Griswold, and D Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. ICSE '99*, pages 213–224, 1999.

[11] L Fan, W Xi, and C Yang. API hyperlinking via structural overlap. In *Proc. ESEC/FSE '09*, pages 203–212, 2009.

[12] Alex Groce and Willem Visser. What went wrong: explaining counterexamples. In *Proc. SPIN'03*, pages 121–136, 2003.

[13] D Jeffrey, N Gupta, and R Gupta. Fault localization using value replacement. In *ISSTA '08*, pages 167–178, 2008.

[14] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proc. ASE '05*, pages 273–282, 2005.

[15] Miryung Kim. *Analyzing and inferring the structure of code change*. PhD thesis, University of Washington, 2008.

[16] B Lerner, M Flower, D Grossman, and C Chambers. Searching for type-error messages. In *PLDI '07*, pages 425–434, 2007.

[17] B Liblit, M Naik, A X. Zheng, A Aiken, and M I. Jordan. Scalable statistical bug isolation. In *PLDI '05*, 2005.

[18] Ben Liblit. *Cooperative Bug Isolation: Winning Thesis of the 2005 ACM Doctoral Dissertation Competition*, volume 4440 of *Lecture Notes in Computer Science*. Springer, 2007.

[19] LOCC home. http://csdl.ics.hawaii.edu/Plone/research/locc/.

[20] C Pacheco and M D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005*, 2005.

[21] C Pacheco, S K. Lahiri, M D. Ernst, and T Ball. Feedback-directed random test generation. In *ICSE '07*, 2007.

[22] David Roach, Hal Berghel, and John R. Talburt. An interactive source commenter for prolog programs. In *Proc. SIGDOC'90*, pages 141–145, 1990.

[23] Pierre N. Robillard. Schematic pseudocode for program constructs and its computer automation by schemacode. *Commun. ACM*, 29:1072–1089, November 1986.

[24] G Sridhara, E Hill, D Muppaneni, L Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proc. ASE'10*, pages 43–52, 2010.

[25] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proc. ICSE '11*, May 25–27, 2011.

[26] W Visser, C S. Păsăreanu, and S Khurshid. Test input generation with Java PathFinder. In *ISSTA '04*, pages 97–107, 2004.

[27] Mark Weiser. Program slicing. In *Proc. ICSE '81*, 1981.

[28] T Xie, D Marinov, W Schulte, and D Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS 2005*, pages 365–381, 2005.

[29] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28:183–200, February 2002.

[30] Sai Zhang. Palus: a hybrid automated test generation tool for Java. In *ICSE '11 SRC track*, pages 1182–1184, 2011.

[31] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. Combined static and dynamic automated test generation. In *ISSTA '11*, pages 353–363, 2011.