Practical Semantic Test Simplification

Sai Zhang
University of Washington, USA
szhang@cs.washington.edu

***Abstract*—We present a technique that simplifies tests at the *semantic* level. We first formalize the *semantic test simplification* problem, and prove it is NP-hard. Then, we propose a heuristic algorithm, SimpleTest, that automatically transforms a test into a simpler test, while still preserving a given property. The key insight of SimpleTest is to *reconstruct* an executable and simpler test that exhibits the given property from the original one. Our preliminary study on 7 real-world programs showed the usefulness of SimpleTest.**

I. Introduction

A test must be effective at detecting bugs, but it must be easy to understand. A test that finds no bug is useless for testing. However, a bug-revealing test requiring substantial comprehension effort, will become less useful in practice.

Recently, a number of techniques have been developed to automatically *generate* new tests [5], [10]; however, there is few solution to *simplify* an existing test to assist programmers in understanding its behavior. In most scenarios, not every statement in a test is equally important to a programmer. In particular, when a test fails by violating a predefined assertion, programmers may only be interested in the test code that is related to the assertion violation, and ignore other irrelevant parts. One practical solution is to provide programmers a *simplified* but still *executable* test that requires less comprehension effort. The simplified test violates the same assertion, but by much shorter sequence. Programmers can choose to inspect the simplified version to understand its failure cause. Besides helping programmers diagnose a bug, a simplified test takes less time and fewer resource to execute and is easier to explain and communicate between programmers.

Existing approaches. Two existing approaches to simplify a test are Delta debugging [8] and program slicing [7]. Delta debugging [3], [8], [9], a general technique to simplify failure-inducing inputs, takes a set of factors (i.e., all statements in a failed test) that might influence a test outcome, and repeats the test with subsets of these factors. By keeping only those factors that are relevant to the test outcome, it systematically reduces the set of factors until a minimized set is obtained. Program slicing [1], [4], [7] uses control/data dependence information to identify a subset of statements that may affect a given property in a program. However, both Delta debugging and program slicing are *fundamentally* limited in simplifying a test. They essentially simplify a test exclusively at the *syntax* level by isolating a statement *subset*, but are ineffective to simplify an *already syntactically-minimized* test, which can still contain many irrelevant statements.

Figure 1 shows a syntactically-minimized failed test, in which if any statement is removed, the test would either be uncompilable or no longer fail. Delta debugging can no longer simplify this test, and computing a backward slice from the failing assertion (line 10) still does not help: the slice consists of all statements in Figure 1. In fact, the test in Figure 1 reveals that the `add` method in the `TreeSet` class is buggy: it should not accept a non-comparable object, but it does (line 8). Code in lines 2 – 6 is entirely irrelevant about this bug, and should be removed without distracting programmers' attention.

```
public void test1() {
1.   Object var1 = new Object();
2.   Integer var2 = 0;
3.   Integer var3 = 1;
4.   Object[] var4 = new Object [] {var1, var2, var3};
5.   List var5 = Arrays.asList(var4);
6.   List var6 = var5.subList(var2, var3);
7.   TreeSet var7 = new TreeSet();
8.   boolean var8 = var7.add(var6);
9.   Set var9 = Collections.synchronizedSet(var7);
      //This assertion fails
10.  assertTrue(var9.equals(var9));
}
```

Fig. 1. An automatically-generated test by Randoop [5] that reveals a bug in JDK 1.6. It shows a call sequence ending with the creation of an object that is not equal to itself. This test has already been syntactically-minimized. Existing approaches [3], [4] can not further simplify this test.

Proposed solution. This paper presents SimpleTest, a test simplification technique by exploring the space of simplifying tests at the *semantic* level. Given a failed test that satisfies a property ϕ, SimpleTest automatically transforms it into a simpler test that still exhibits ϕ. To efficiently achieve this, SimpleTest greedily computes a locally optimal simplification instead of the globally optimal simplification. A key difference of SimpleTest from existing test simplification techniques [3], [4] is that SimpleTest tries to *reconstruct* an executable test from the original one, instead of *carving* a subset of the existing test code. SimpleTest starts from ϕ, follows back data dependencies, and repeatedly *replaces* referred expressions in each statement with other alternatives from the test code itself. After each replacement, SimpleTest constructs a simpler test. The simpler test is immediately executed to validate whether ϕ is still satisfied. SimpleTest repeats the above replacement process until the result test can no longer be simplified. As an example, Figure 2 shows a simplified test for the failed test in Figure 1. This simplified test triggers the same assertion at line 10, but is much shorter than the original one. In Figure 2, a crucial step conducted by SimpleTest is to replace the referred variable `var6` at line 8 with the variable `var1` defined at line 1. This replacement opens the possibility to further simplify the test by removing irrelevant statements from line 2 to 6,

978-1-4673-3076-3/13/$31.00 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA
New Ideas and Emerging Results

```
public void test1() {
1.   Object var1 = new Object();
2.   Integer var2 = 0;
3.   Integer var3 = 1;
4.   Object[] var4 = new Object [] {var1, var2, var3};
5.   List var5 = Arrays.asList(var4);
6.   List var6 = var5.subList(var2, var3);
7.   TreeSet var7 = new TreeSet();
8.   boolean var8 = var7.add(var1); //it was var6
9.   Set var9 = Collections.synchronizedSet(var7);
      //This assertion fails
10.  assertTrue(var9.equals(var9));
}
```

Fig. 2. A simplified test by our SimpleTest algorithm for the test in Figure 1. Line numbers are aligned with Figure 1. The ~~strikeout~~ part is not shown in the final simplified test, and the replaced variable during simplification is highlighted by **underline**.

and yields a much simpler test (the code in Figure 2 without the ~~strikeout~~ lines).

Contributions. The main contributions of this paper are:

- **Problem.** We introduce the problem of *semantic test simplification* and prove its NP-hardness (Section II).
- **Technique.** We propose SimpleTest, a simple but effective test simplification algorithm (Section III).
- **Evaluation.** We perform a preliminary study to show the usefulness of SimpleTest (Section IV).

II. Formalism

In this section, we first formalize the semantic test simplification problem, then prove its NP-Hardness.

Definition 1. *A unit test t consists of a sequence of statements $\overline{s}$ = $\{s_1, s_2, ..., s_n\}$ and a predicate ϕ as a testing oracle.*

We write $t \rightarrow \phi$ when the execution of $\overline{s}$ satisfies ϕ. Similarly, we write $t \nrightarrow \phi$ when the execution of $\overline{s}$ does *not* satisfy ϕ[1]. We use $|t|$ to denote the length of a unit test t, which is equal to the number of statements in $\overline{s}$.

Definition 2. *A statement $s \in \overline{s}$ in a unit test t is either a value declaration statement or a method invocation statement:*

$$s ::= \quad v_{out} = v_{in} \\ \mid v_{out} = m(v_{in_1}, ..., v_{in_n})$$

A value declaration statement simply assigns a value v_{in} to a declared variable v_{out}. A method invocation statement invokes method m with argument variables v_{in_1}, v_{in_2}, ..., v_{in_n}, and assigns the result to an output variable v_{out}. In our definition, value declaration statements include class/static/local field assignments, and method invocation statements include constructor calls and array operations. Our definition models *deterministic* and *sequential* test code, and does not cover many sophisticated Java language features (e.g., threading and exception handling).

We use $\perp$ to denote v_{out} if no output is available, such as invoking a `void` method. We define two predicates isValDecl(s) and isMeCall(s) to check whether a statement s is a value declaration statement or a method invocation statement, respectively.

[1]For example, the bug-revealing property ϕ of the failed test in Figure 1 is: `var9.equals(var9)` returns `false` at line 10.

For $s \in \overline{s}$, we use $\mathcal{O}(s)$ to denote its return variable v_{out}, and use $\mathcal{I}(s)$ to denote its input variable set:

$$\mathcal{O}(s) = \begin{cases} \perp & \text{if isMeCall(s) and } \textit{returnType}\text{(m)=}\texttt{void} \\ v_{out} & \text{otherwise} \end{cases}$$

$$\mathcal{I}(s) = \begin{cases} \{v_{in}\} & \text{if isValDecl(s)} \\ \{v_{in_1}, ..., v_{in_n}\} & \text{if isMeCall(s)} \end{cases}$$

For property ϕ, $\mathcal{I}(\phi)$ denotes the input variables to evaluate ϕ. To be a valid test, for each $v_i \in \mathcal{I}(\phi)$, there must $\exists s \in \overline{s}$ such that $v_i = \mathcal{O}(s)$. $\mathcal{O}(\phi)$ denotes the evaluation result of ϕ as:

$$\mathcal{O}(\phi) = \begin{cases} \text{T} & \text{if } t \rightarrow \phi \\ \text{F} & \text{if } t \nrightarrow \phi \end{cases}$$

For each $s \in \overline{s}$, we define:

$$\text{getValue}(s) = \begin{cases} v_{in} & \text{if isValDecl(s)} \\ \perp & \text{otherwise} \end{cases}$$

$$\text{getMethod}(s) = \begin{cases} m & \text{if isMeCall(s)} \\ \perp & \text{otherwise} \end{cases}$$

For two statements $s, s' \in \overline{s}$, we define:

SameStatementKind(s, s') =

$$\begin{cases} \text{T} & \text{if isValDecl}(s) == \text{isValDecl}(s') \\ & \quad \&\& \text{ isMeCall}(s) == \text{isMeCall}(s') \\ \text{F} & \text{otherwise} \end{cases}$$

SameValueOrMethod(s, s') =

$$\begin{cases} \text{T} & \text{if SameStatementKind}(s, s') \\ & \quad \&\& \text{ getValue}(s) == \text{getValue}(s') \\ & \quad \&\& \text{ getMethod}(s) == \text{getMethod}(s') \\ \text{F} & \text{otherwise} \end{cases}$$

Definition 3. *A unit test $t = \overline{s} \cup \phi$ is valid iff:*
(1). each statement $s \in \overline{s}$ is type-correct
(2). $\forall\ s_i \in \overline{s}$, and $\forall\ v \in \mathcal{I}(s_i)$, there $\exists\ s_j \in \overline{s}$ with $j < i$, such that $v = \mathcal{O}(s_j)$.

Definition 4. *Given a valid unit test $t = \overline{s} \cup \phi$ and $t \rightarrow \phi$, $t' = \overline{s'} \cup \phi'$ is a simplified test of t if t' satisfies:*
(1). $\phi' = \phi$ and $t' \rightarrow \phi'$
(2). t' is valid, and $\forall\ s'_i \in \overline{s'}$, there $\exists\ s \in \overline{s}$, such that SameValueOrMethod(s'_i, s) returns true
(3) $\forall\ s'_i, s'_j \in \overline{s'}$ with $i < j$, there $\exists\ s_k, s_p \in \overline{s}$ with $k < p$, such that both SameValueOrMethod(s'_i, s_k) and SameValueOrMethod(s'_j, s_p) return true.

In our definition, a simplified test does *not* invoke new methods, re-order existing statements, or use new values. Instead, all input variables are *reused* from the original test. This tends to avoid changing the testing intention of the original test code and prevent increasing test code complexity.

We next define the *semantic test simplification* problem.

Definition 5. *Given a valid test $t = \overline{s} \cup \phi$ and $t \rightarrow \phi$, find a simplified test t_{min}, such that: for any t' that is also a simplified test of t, and $|t'| \geq |t_{min}|$.*

SimpleTest
Input: a unit test $t = \overline{s} \cup \phi$
Output: a simplified test $t' = \overline{s'} \cup \phi$
Auxiliary Methods:
getReconstructionIndex(t, s_i, j): returns the the index of the first statement in test t whose output variable is type-compatible with the j-th input variable of statement s_i.
reverseStmtOrder(t): returns all statements in a test t in reverse order.
Invariants:
checkProperty(t, ϕ) = true
checkProperty(t', ϕ) = true

```
1:  t' ← removeRedundantStmts(t, φ)
2:  for each statement s_i in reverseStmtOrder(t') do
3:    for each input j in statement s_i do
4:      replace_index ← getReconstructionIndex(t', s_i, j)
5:      t_s ← reconstruct(t', s_i, j, s_replace_index)
6:      if checkProperty(t_s, φ) then
7:        t' ← removeRedundantStmts(t_s, φ)
8:      end if
9:    end for
10: end for
11: return t'
```

Fig. 3. The SimpleTest algorithm for semantically simplifying a test.

The goal is to find a simplified test t_{min} that has the shortest length among all simplified tests and still satisfies ϕ. Such a test minimizes the number of statements, which, in our view, is simpler than the original test.

Unfortunately, the problem of finding an element of t_{min} is NP-hard. We next prove this claim by reducing the Set Cover problem [2] to the test simplification problem.

THEOREM 1. *The problem of finding an element of t_{min} is NP-hard.*

Proof. (Sketch) In the Set Cover problem, we are given a set of elements $U = \{c_1, ..., c_m\}$ (called the universe), and n sets $S_1, ..., S_n$ whose union comprises the universe. The goal is to identify the *smallest* number of sets from $S_1, ..., S_n$ whose union still contains all elements in the universe. The Set Cover problem is well known to be NP-hard [2].

We now reduce the Set Cover problem to the semantic test simplification problem. Assume we are given an arbitrary universe $U = \{c_1, ..., c_m\}$ and n sets $S_1, ..., S_n$ on which we must find the minimum number of sets whose union comprises the universe U. We now construct a test $t = \overline{s} \cup \phi$ as follows:

- $\overline{s}$ consists of $n+m$ statements. The first n statements are value declaration statements, and the next m statements are non-`void` method invocation statements. For convenience, we denote n value declaration statements as: s_{v_1}, ..., s_{v_n}, and denote m method invocation statements as: s_{c_1}, ..., s_{c_m}.
- Each of the m method invocation statements takes exactly one input variable, and returns one output variable. That is, $|\mathcal{I}(s_{c_i})| = |\mathcal{O}(s_{c_i})| = 1$. For each set S_i, if element $c_j \in S_i$, we denote the output variable of statement s_{v_i} can be used as the input variable of statement s_{c_j}. That is, s_{v_i}'s output variable is type-compatible with s_{c_j}'s input variable.
- Predicate ϕ uses *all* output variables of the m method invocation statements. That is, $|\mathcal{I}(\phi)| = m$, and for each s_{c_i} with $1 \leq i \leq m$, $\mathcal{O}(s_{c_i}) \in \mathcal{I}(\phi)$.

In the above construction, m method invocation statements can not be removed, since ϕ uses all their output variables. Thus, in any simplified test $t' = \overline{s'} \cup \phi$, $s_{c_i} \in \overline{s'}$ ($1 \leq$ i $\leq m$).

Let $t_{min} = \overline{s_{min}} \cup \phi$ be a minimally simplified test. For each $1 \leq$ i $\leq n$, if $s_{v_i} \in \overline{s_{min}}$, we add set S_i to the result. Since the size of t_{min} has been minimized, this leads to a minimal number of sets in the result. This shows that the test simplification problem is NP-hard. □

III. THE SimpleTest Algorithm

Given a failed test $t = \overline{s} \cup \phi$, where $\overline{s} = s_1, ..., s_n$, SimpleTest has three basic actions on t to simplify it:

- *reconstruct(t, s_i, j, s_k)*: returns a new test by replacing the j-th input variable of statement s_i with the output variable of statement s_k. When applying this operation, s_i must be a method invocation statement having at least j input variables, and the output variable of the k-th statement must not be $\perp$.
- *removeRedundantStmts(t, ϕ)*: returns a syntactically-minimized test of t that still exhibits ϕ. This action performs syntax-level test simplification by removing all redundant statements which will not affect the test execution result. This action can be instantiated using any syntax-level test simplification technique. Our current implementation repeatedly removes each statement in t until no more statements can be removed. After a statement is removed, our implementation immediately executes the result test to validate whether ϕ is still satisfied.
- *checkProperty(t, ϕ)*: returns `true` if $t \rightarrow \phi$, and `false` if $t \nrightarrow \phi$. In our implementation, ϕ is a Java assertion.

The SimpleTest algorithm is described in Figure 3. It first syntactically simplifies a given test by performing the *removeRedundantStmts* action (line 1). Then, it *greedily* reconstructs each statement by replacing its input variables with possible alternatives created *earlier* in the same test (line 4). After each reconstruction, the algorithm first checks whether ϕ is still satisfied (line 6). If the reconstructed test still exhibits ϕ, SimpleTest performs the *removeRedundantStmts* action again to conduct further syntax-level test minimization (line 7). Otherwise, it rolls back to the test before reconstruction and continues to replace the next input variable (line 3).

THEOREM 2. *The worst-case time complexity of* SimpleTest is $\mathcal{O}(n^3)$, where n is the number of statements in a test.

Proof. (Sketch) Each of the for loop in Figure 3 has an upper bound of n iterations. In our implementation, the worst-case time complexity of the *removeRedundantStmts* method is $\mathcal{O}(n)$. Therefore, the cost of the entire SimpleTest algorithm is $\mathcal{O}(n^3)$

Subject Programs		Failed Tests			Static Slicing		Delta Debugging [3]		DD + Slicing [4]		**SimpleTest**	
Name (version)	Lines of Code	Tests	Bugs	Stmts	Stmts	Time	Stmts	Time	Stmts	Time	Stmts	Time
Time And Money (0.51)	2372	23	3	1337	1337	0.1	1032	68.6	1032	70.0	90	20.6
jdom (1.1)	8513	3	1	194	194	0.1	93	16.8	93	17.2	19	8.5
Apache Commons Primitives (1.0)	9368	5	2	377	377	0.1	142	23.8	142	26.2	37	20.5
Apache Commons Beanutils (1.8.3)	11382	10	2	317	317	0.1	166	10.4	166	10.1	53	6.5
Apache Commons Math (2.2)	14469	18	3	747	705	0.1	490	35.8	467	33.5	114	19.3
Apache Commons Collections (3.2.1)	55400	8	3	399	399	0.1	283	100.9	283	105.8	49	59.1
`java.util` package (1.6.0_12)	48026	6	2	178	178	0.1	107	10.2	107	10.4	36	6.6
Total	149557	73	16	3549	3507	<1s	2313	266.5	2290	273.2	398	141.1

Fig. 4. Experimental results. Column "Failed Tests" shows the information of the original failed tests. Sub-column "Tests" shows the number of failed tests, sub-column "bugs" shows the number of distinct bugs revealed, and sub-column "Stmts" shows the total number of test statements (excluding assertions). Column "Static Slicing" shows the results of using static slicing for test simplification. Column "Delta debugging" shows the results of using a Delta debugging-based test simplification technique [3]. Column "DD + Slicing" shows the results of using an existing approach by combining static slicing and Delta debugging [4]. Column "**SimpleTest**" shows the results by using the SimpleTest algorithm proposed in this paper. In each column, sub-column "Stmts" shows the number of statements *after* simplification (lower is better), and sub-column "Time" shows the time cost in seconds.

IV. Preliminary Evaluation

We implemented the SimpleTest algorithm in a prototype tool. Our tool takes as input a JUnit test with a predicate in the form of Java assertion. It parses the JUnit test, conducts semantic test simplification, and outputs a simplified test that still satisfies the same predicate.

We evaluated SimpleTest's effectiveness by simplifying 73 bug-revealing unit tests from 7 real-world programs (see the "Subject Programs" column in Figure 4). The tests are generated by Randoop [5]; each of which reveals a real bug. We also compared SimpleTest's effectiveness with three existing syntax-level test simplification techniques, namely, (intra-procedure) static slicing [7], Delta debugging [3], [8], and the combination of Delta debugging and static slicing [4].

As indicated by Figure 4, SimpleTest is surprisingly effective: it reduces the size of 73 failed tests from 3549 statements to 398 statements code in 141.1 seconds.

For each failed test, we manually wrote an optimally-simplified test (i.e., the shortest sequence that revealed the same defect) to check whether SimpleTest can produce the optimal result. As a result, 72 out of 73 simplified failed tests are optimally simplified; only 1 test from Apache Commons Beanutils is sub-optimal (the optimal test size is 6 while SimpleTest outputs a test with size 7). This slight difference is caused by SimpleTest's greedy heuristic when performing code transformation.

Compared to existing approaches [3], [4], [7], [8], the simplified tests output by SimpleTest are substantially smaller than the output by other approaches (8.8X smaller than the results of static slicing, and about 5.5X smaller than the results of Delta debugging and the combination of Delta debugging with static slicing). It uses a moderate amount of time (slower than static slicing, but about 1.9X faster than Delta debugging and the combination of Delta debugging with static slicing).

V. Conclusion and Future Work

In this paper, we introduced and modelled the problem of *semantic test simplification*, proved its NP-hardness, and proposed a simple but effective algorithm, SimpleTest, that often yields the optimal solution. Our preliminary evaluation showed the effectiveness of SimpleTest.

For future work, we plan to extend SimpleTest to handle test code containing if-else conditions and loops, and conduct a user study to understand how a semantically-simplified test helps programmers diagnose a bug. We are also interested in comparing SimpleTest with other test simplification an understanding techniques (e.g., dynamic slicing [1], C-Reduce [6], FailureDoc [11], and AutoFlow [9]), and exploring potential downstream applications of SimpleTest.

Acknowledgements

This work was supported in part by ABB Corporation and NSF grant CCF-1016701.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, 1990.
[2] J. Kleinberg. *Algorithm Design*. Pearson Education, 2006.
[3] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *ISSRE*, pages 267–276, 2005.
[4] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *ASE*, 2007.
[5] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE '07*, pages 75–84.
[6] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for c compiler bugs. In *PLDI*, 2012.
[7] M. Weiser. Program slicing. In *ICSE*, 1981.
[8] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *TSE*, 28:183–200, February 2002.
[9] S. Zhang, Y. Lin, Z. Gu, and J. Zhao. Effective identification of failure-inducing changes: a hybrid approach. In *PASTE*, 2008.
[10] S. Zhang, D. Saff, Y. Bu, and M. D.Ernst. Combined static and dynamic automated test generation. In *ISSTA' 2011*.
[11] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *Proc. ASE '11*.