DOI:10.1145/2856103

Satisfiability modulo theory solvers can help automate the search for the root cause of observable software errors.

BY ABHIK ROYCHOUDHURY AND SATISH CHANDRA

Formula-Based Software Debugging

PROGRAMMING, THOUGH A creative activity, poses strict demands on its human practitioners in terms of precision, and even talented programmers make mistakes. The effect of a mistake can manifest in several ways—as a program crash, data corruption, or unexpected output. Debugging is the task of locating the root cause of an error from its observable manifestation. It is a challenge because the manifestation of an error might become observable in a program's execution much later than the point at which the error infected the program state in the first place. Stories abound of heroic efforts required to fix problems that cropped up unexpectedly in software that was previously considered to be working and dependable.

Given the importance of efficient debugging in overall software productivity, computer-assisted techniques for debugging are an active topic of research. The premise of such techniques is to employ plentiful compute cycles to automatically narrow the scope of where in the source code the root cause is likely to be, thereby reducing the amount of time a programmer must spend on debugging. Such computer-assisted debugging techniques, as discussed in this article, do not promise to pinpoint the mistake in a program but only to narrow the scope of where the mistake might lurk. Such techniques are also sometimes called "fault localization" in the software-engineering literature.

We now proceed to review the major advances in computer-assisted debugging and describe one of the major challenges in debugging—lack of specifications capturing the intended behavior of the program; that is, if the intended behavior of the program is not available, how can a debugging method infer where the program went "wrong"? We next present a motivating example extracted from Address Resolution Protocol (ARP) implementation from GNU Coreutils[10] that serves as a running example in the article.

We then discuss at a high level how symbolic techniques can help in this direction by extracting candidate specifications. These techniques utilize some form of symbolic execution, first introduced by King.[14] We later describe debugging techniques that use some form of symbolic techniques to recover specifications. Table 1 outlines

» key insights

- **The lack of explicit formal specifications of correct program behavior has long held back progress in automated debugging.**
- **To overcome this lack of formal specifications, a debugging system can use logical formula solving to attempt to find the change in a program that would make the failed test pass.**
- **Because the failure of a test in a program can also be seen as an unsatisfiable logical formula, debugging—the task of explaining failures—can thus benefit from advances in formula solving or constraint satisfaction.**

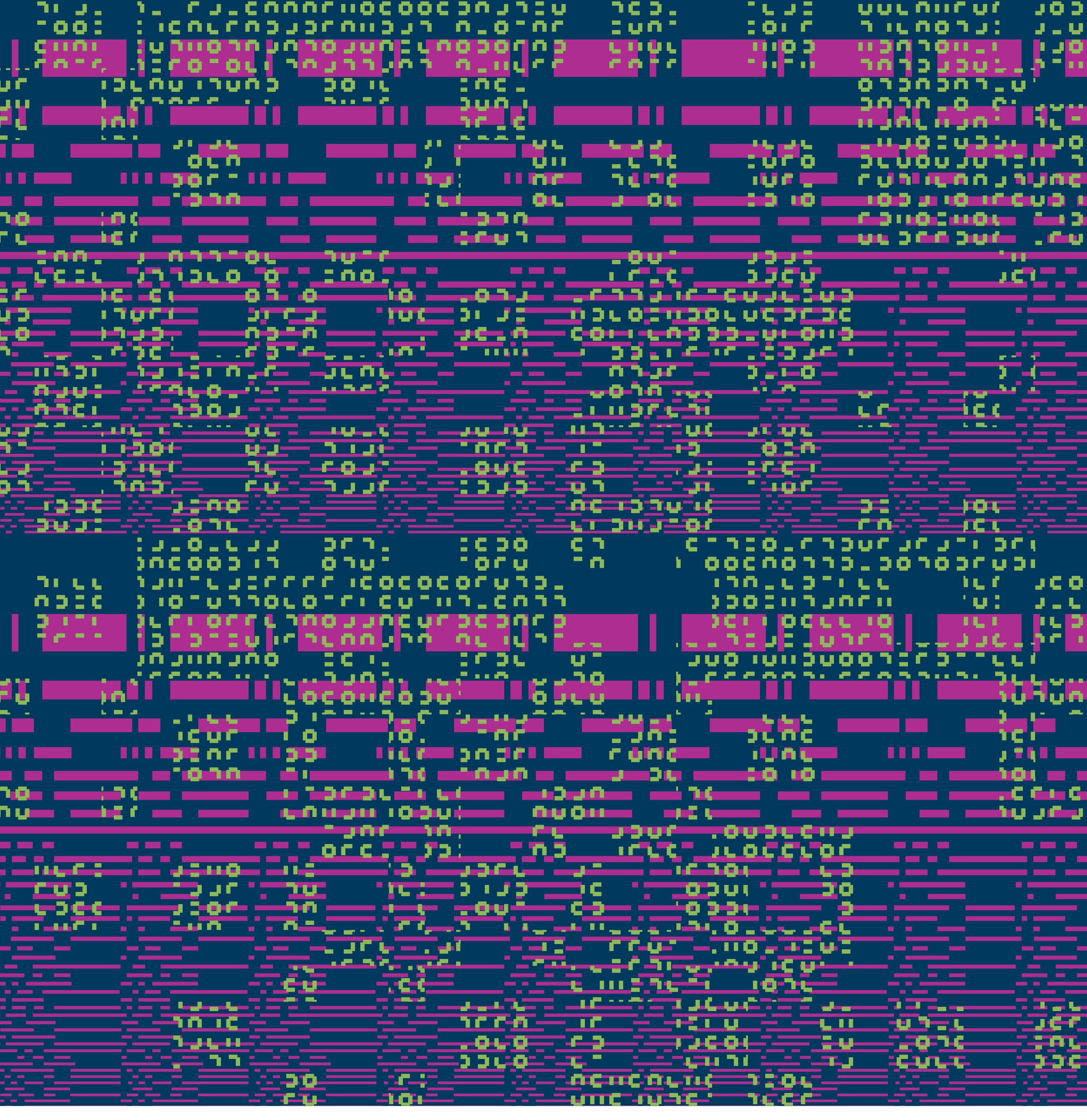


IMAGE BY CEPERA

the symbolic techniques we discuss and their relation to indirect specifications. These techniques all perform symbolic analysis of various program artifacts (such as failing traces or past program versions) to help guide a programmer's search for possible causes of an observable error. Program artifacts are converted into a logical formula through symbolic analysis, and manipulation of such logical formulae helps uncover specifications of intended program behavior. We conclude with a forward-looking view of symbolic analysis used for automated program repair, as reflected in research projects DirectFix,[17] Angelix,[18] and SemFix.[20]

Computer-Assisted Debugging

There has been interest in computer-assisted debugging since at least the mid-1980s. Here, we highlight three major ideas and refer the reader to Zeller[25] for a more comprehensive overview of these and other ideas in computer-assisted debugging.

The first major idea in harnessing the power of computers to aid programmers in debugging was "slicing." Static slicing[23] aims to preserve only as much of the program as is necessary to retain the behavior as far as some output variable is concerned; such

controlled preservation is achieved through a static approximation of control and data dependencies. Dynamic analysis[1] applies the same idea but to an execution of the program on a specific input; the advantage is only that the control and data dependencies in the execution are used in the computation of the slice, leading to a more succinct and precise slice.

The second significant idea is "delta debugging"[7,24] in which a programmer tries to isolate the cause of a failure by systematically exploring deviations from a non-failure scenario. For example, if a new version of code breaks while the old version works, one can systematically try to isolate the specific change in the program that can be held responsible for the failure; the same idea also applies to program executions. Delta debugging takes advantage of compute cycles by systematically exploring a large number of program variations.

The third idea we highlight is "statistical fault isolation,"[12,15] which looks at execution profiles of passing and failing tests. If execution of a statement is strongly correlated (in a statistical sense) with only the failing tests, it is ranked highly in its suspiciousness.

Such ideas shift the burden of localizing an observable error from programmer to computer. Techniques like delta debugging rely on exploration or search over inputs or over the set of states in a trace to localize the cause of error.

Note the debugging problem implicitly contains search-based subproblems (such as the locations at which the program could be altered to avoid the observable error or which successful trace in the program you can choose to compare a given failing trace). These search problems in the debugging methods outlined earlier would be solved through various search heuristics. In contrast, the symbolic analysis-based debugging methods we present here solve these search problems by "solving logical formulae." This new category of methods has emerged essentially out of an opportunity—the maturity and wide availability of satisfiability modulo theory (SMT) solvers.[8] SMT formulae are in first-order logic, where certain symbols appearing in the formula come from background theories (such as theory of integers, real numbers, lists, bitvectors, and arrays). Efficient solving of SMT formulae allows us to logically reason about programs (static analysis) or about program artifacts like traces (dynamic analysis). The maturity of SMT solvers has made symbolic execution more practical, triggering new directions in software testing,[4] where symbolic execution and SMT constraint solving are used to guide the search over the huge search space of program paths for generating tests. In this article, we show how SMT solvers can be put to use in the automation of software debugging.

Running Example

We now present a running example (see Figure 1) drawn from real-life code. It is a simplified version from a fragment of the ARP implementation in GNU Coreutils[10] in which a bug is introduced when a redundant assignment is added at line 5. We use it to illustrate the various debugging methods explored throughout the article.

There is an assertion in the program that captures a glimpse of the intended behavior. The rest of the intended behavior is captured through the following test cases. Without loss of generality, assume `DEFAULT`, `NULL`, `ETHER`, `INET` appearing in the program and/or test cases are predefined constants.

```
Test 1:
   arp A INET H ETHER (passing test).
   Expected output: ETHER
Test 2:
   arp A INET (failing test).
   Expected output: DEFAULT
   Actual output: NULL (assert fails)
Test 3:
   arp H ETHER (passing test).
   Expected output: ETHER
```

The program has a redundant assignment in line 5 that changes the control flow of execution of `Test 2` but not of the other tests. The violation of the intended behavior in this test is reflected in the failure of the assertion, as well as in observing an output that is different from the expected output.

The question here for a programmer is based on the failure of a test: How can the root cause be found in the failure? Being able to answer depends on a specification of the intended behavior so we can find the root cause of where the program behavior turned incorrect. On the other hand, in most ap-

Table 1. Debugging using symbolic techniques.

Name	Symbolic Technique	Information from
BugAssist[13]	Program Formula	Internal inconsistency
Error Invariants[10]	Interpolants	Internal inconsistency
Angelic Debugging[5]	Static Symbolic Execution	Passing tests
Darwin[22]	Dynamic Symbolic Execution	Previous version

Figure 1. Running example buggy program.

```
0 hw_set = 0; hw = NULL; ap = NULL;
1 while (i = getopt(....)) {
2    switch (i) {
3        case 'A':
4             ap = getaftype(optarg);
5             hw_set = 1;  // BUG: redundant line
6             break;
7        case 'H':
8             hw = gethwtype(optarg);
9             hw_set = 1;
10            break;
11    }
12 }
13 if (hw_set == 0) {
14          hw = gethwtype(DEFAULT);
15 }
16 assert(hw  != NULL);
17 printf("%d", hw);
```

plication domains, programmers do not write down detailed specifications of the intended behavior of their code. Protocol specifications, even when available, are usually at a much higher level than the level of the implementation described here. In this particular case, when `Test 2` fails, how does the programmer infer where the program execution turned incorrect?

The execution trace of `Test 2` is as follows

```
0 hw _ set = 0; hw = NULL; ap = NULL;
1 while (i = getopt(....)) {
2    switch (i) {
3         case 'A':
4              ap = getaftype(optarg);
5              hw _ set = 1;
6              break;
11   } // exit switch statement
1 while (i = getopt(....)) {
12 } // exit while loop
13 if (hw _ set == 0)
{ // this condition is false
16 assert(hw != NULL);
// assertion fails
```

So, when the assertion fails in line 16, which line in the program does the programmer hold responsible for this observed error? Is it line 13, where the condition checked could have been different (such as `hw set == 1`)? If this was indeed the condition checked in line 13, `Test 2` would not fail. Is it line 5, where `hw _ set` is assigned? This is the line we hypothesized as the bug when we presented the buggy code to a human, but how does a computer-aided debugging method know which line is the real culprit for the observed error, and can it be fixed?

In general, for any observable error, there are several ways to fix a fault, and the definition of the fault often depends on how it is fixed. Since specifications are unavailable, have we thus reached the limit of what can be achieved in computer-assisted debugging? Fortunately, it turns out some notion of intended behavior of the program can be recovered through indirect means (such as internal inconsistency of the failed trace, passing test cases, or an older working version of the same program). In this article, we discuss debugging methods that rely on such indirect program specifications to find the root cause of an observable error.

Fortunately, it turns out some notion of intended behavior of the program can be recovered through indirect means.

Using Satisfiability

The technique we discuss first is called "BugAssist"[13] in which the input and desired output information from a failing test are encoded as constraints. These input-output constraints are then conjoined with the program formula, which encodes the operational semantics of the program symbolically along all paths; see Figure 2 for an overview of conversion from a program to a formula. In the example program of Figure 1, we produce the following formula (ϕ):

$$
\begin{aligned}
\phi = & \; arg[1] = A \wedge arg[2] = INET \wedge arg[3] = NULL \\
& \wedge hw_set_0 = 0 \wedge hw_0 = NULL \wedge ap_0 = NULL \\
& \wedge i_1 = arg[1] \wedge i_1 \neq NULL \\
& \wedge guard_3 = (i_1 == A) \\
& \wedge ap_4 = arg[2] \\
& \wedge hw_set_5 = 1 \\
& \wedge ap_{11} = guard_3 \; ? \; ap_4 : ap_0 \\
& \wedge hw_set_{11} = guard_3 \; ? \; hw_set_5 : hw_set_0 \\
& \wedge i'_1 = arg[3] \wedge i'_1 == NULL \\
& \wedge guard_{13} = (hw_set_{11} == 0) \\
& \wedge hw_{14} = DEFAULT \\
& \wedge hw_{15} = guard_{13} \; ? \; hw_{14} : hw_0 \\
& \wedge hw_{15} \neq NULL \wedge hw_{15} == DEFAULT
\end{aligned}
$$

The *arg* elements refer to the input values, similar to *argv* of the C language; here, the inputs are for `Test 2`. The last line of clauses represents the expectation of `Test 2`. The remainder of the formula represents the program logic. (For brevity, we have omitted the parts of the formula corresponding to the `case 'H'`, as it does not matter for this input.) The variable i'_1 refers to the second time the loop condition `while` is evaluated, at which point the loop exits. We use = to indicate assignment and == to indicate equality test in the program, though for the satisfiability of the formula both have the same meaning.

The formula ϕ, though lengthy, has one-to-one correspondence to the trace of `Test 2` outlined earlier. Since the test input used corresponds to a failing test, the formula is unsatisfiable.

The BugAssist tool tries to infer what went wrong by trying to make a large part of the formula satisfiable, accomplishing it through MAX-SAT[19] or MAX-SMT solvers. As the name suggests, a MAX-SAT solver returns the largest possible satisfiable sub-formula of a formula; the sub-formula omits

Figure 2. Conversion of program to formula; the formula encodes, using guards, possible final valuations of variables.

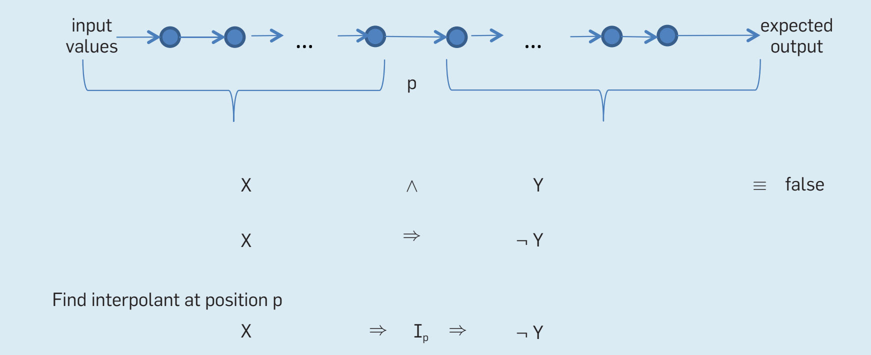
```
1 input y; // initially x = z = 0
2 if (y > 0){
3     z = y * 2;
4     x = y - 2;
5     x = x - 2; }
6 if (z == x)
7     output("How did I get here");
8 else if (z > x)
9     output("Error");
```

Here is the corresponding formula

$guard_2 = (y > 0)$
$\wedge\ z_3 = y * 2$
$\wedge\ x_4 = y - 2$
$\wedge\ x_5 = x_4 - 2$
$\wedge\ z_6 = guard_2\ ?\ z_3 : 0$
$\wedge\ x_6 = guard_2\ ?\ x_5 : 0$
$\wedge\ guard_6 = (z_6 == x_6)$
$\wedge\ guard_8 = (z_6 > x_6)$
$\wedge$ *output* = *guard*$_6$? `How ...` : (guard$_8$? `Error:` nil)

In it, variables are given a subscript based on the line on which an instance is assigned. *Guard* variables denote conditions that regulate values of variables when potentially different values of a variable reach a branch point. For example, $guard_2$ regulates the value of z_6 based on whether the default initial value or the assignment to z_3 reaches it.

Figure 3. Using interpolants to analyze error traces.



certain conjuncts of the original formula. Moreover, the MAX-SAT solver is instructed that certain constraints are *hard* constraints, in that they cannot be omitted. Constraints on input and output are typically hard. The solvers will find the maximum part of the formula, which is satisfiable, thereby suggesting the minimal portion of the program that needs to be changed to make the formula satisfiable. In this sense, the technique attempts to use internal inconsistency to help a programmer, as an indirect source of specification.

We now illustrate how BugAssist would work on the aforementioned formula. We first mark the clauses related to *args*, as well as the final constraints on hw_{15} as hard. The MAX-SAT solver could first highlight the constraint $guard_{13} = (hw_set_{11} == 0)$ as the one to be omitted for making the rest of the formula satisfiable. The reader can verify that setting the (now) unbound variable $guard_{13}$ to true will make hw_{15} equal to hw_{14}, satisfying the output constraints. In terms of the program, this corresponds to an attempt to fix the program at line 13

```
13: if (hw _ set == 1) { // FIX: changed guard
```

Even though this fix passes `Test 2`, it will fail previously passing tests like `Test 1`, thereby introducing regressions. The fix is thus incorrect. However, BugAssist does not vet any of the code highlighted by the technique, relying instead on the programmer to assess whether the suggested constraint corresponds to a good fix. Realizing this potential regression, the programmer using BugAssist would mark $guard_{13} = (hw_set_{11} == 0)$ as a hard constraint. Marking as a hard constraint indicates to the solver it should explore other lines in the program as possible causes of error.

The MAX-SAT solver will then highlight another (set of) constraints, say, $hw_set_5 = 1$, meaning just this constraint can be omitted (or changed) to make the overall formula satisfiable. The reader can verify this forces $guard_{13}$ to be true. This corresponds to identifying line 5 in Figure 1 as the place to be fixed. Here is the correct fix

```
5: // hw _ set = 1; FIX: deleted line
```

Interestingly, from the perspective of the satisfiability of the formula, changing the value assigned to `hw _ set` from 1 to 0 is also a plausible but not robust fix, meaning it can fail other tests, as we show later in the article.

The BugAssist technique tries to extract the reason for failure through analysis of the error trace. Extraction is done iteratively, by successively finding minimal portions of the formula, the omission or alteration of which can make the error trace formula satisfiable. In some sense, the complement of MAX-SAT reported by BugAssist in repeated iterations form legitimate explanations of the observed failure in the error trace being examined. As may be observed even from our simple example, the technique may report several potential faults. It is thus not so much as a one-shot fault-localization method as it is an "iterative exploration" of the potential locations where a change could avert the error in question. The iterative exploration is guided by the computation of the maximum satisfiable portion of an unsatisfiable formula that captures the program failure.

Using Interpolants

An alternative method, called "error invariants,"[9] tries to find the reason for failure by examining error propagation in the different positions of the error trace. Identifying error root-cause is achieved by computing interpolant formula at each position of the error trace. The notion of interpolant[16] requires some explanation. Given a logical implication $X \Rightarrow Y$ involving first-order

logic formulae X and Y, an interpolant is a formula I satisfying

$$X \Rightarrow I \Rightarrow Y$$

The formula I is expressed through the common vocabulary of X and Y. Figure 3 outlines the use of interpolants for analyzing error traces. The error trace formula is a logical conjunction of the input valuation, the effect of each program statement on the trace, and the expectation about output.

Given any position p in the trace, if we denote the formula from locations prior to p as X, and the formula from locations in the trace after p as Y, clearly $X \wedge Y$ is false. Thus $\neg(X \wedge Y)$ holds, meaning $\neg X \vee \neg Y$ holds, meaning $X \Rightarrow \neg Y$ holds. The interpolant I_p at position p in the error trace will thus satisfy $X \Rightarrow I_p \Rightarrow \neg Y$. Such an interpolant can be computed at every position p in the error trace for understanding the reason behind the observable error.

Let us now understand the role of interpolants in software debugging. We first work out our simple example, then conceptualize use of the logical formula captured by interpolant in explaining software errors. In our running example, the interpolants at the different positions are listed in Table 2, including interpolant formula after each statement. Note there are many choices of interpolants at any position in the trace, and we have shown the weakest interpolant in this simple example. The trace here again corresponds to the failing execution of `Test 2` on the program in Figure 1, and we used the same statements earlier in this article on BugAssist.

What role do interpolants play in explaining an error trace? To answer, we examine the sequence of interpolants computed for the error trace in our simple example program, looking at the second column in Table 2 and considering only non-repeated formulae:

$arg[1] = A$
$arg[1] = A \wedge hw_0 = NULL$
$i_1 = A \wedge hw_0 = NULL$
$guard_3 = true \wedge hw_0 = NULL$
$guard_3 = true \wedge hw_0 = NULL \wedge hw_set_5 = 1$
$hw_0 = NULL \wedge hw_set_{11} = 1$
$hw_0 = NULL \wedge guard_{13} = false$
$hw_{15} = NULL$

The sequence of interpolants here shows the propagation of the error via the sequence of variables arg[1], hw_0, i_1, and so on. Propagation through both data and control dependence is tracked. Propagation through data dependence corresponds to an incorrect value being passed from one variable to another through assignment statements. Propagation through control dependence corresponds to an incorrect valuation of the *guard* variables, leading to an incorrect set of statements being executed. Both types of propagation are captured in the interpolant sequence computed over the failed trace. The interpolant at a position p in the error trace captures the "cause" for failure expressed in terms of variables that are live at p. Computing the interpolant at all locations of the error trace allows the developer to observe the error-propagation chain.

The other important observation to make is program statements that do not alter the interpolant are irrelevant to the explanation of the error. In Table 2, statements marked with a • are not relevant to explaining the failure of `Test 2`. For example, anything relevant only to the computation of *ap* is ignored, an approach similar to backward dynamic slicing, though an interpolation-based technique is more general than dynamic slicing. The remaining statements form a minimal error explanation. Once again, the technique uses the internal inconsistency of the faulty execution to figure out possible causes of error. Note, choosing interpolants must be done with care—as interpolants in general are not unique—for the method to be effective in filtering away irrelevant statements.[6,9] Furthermore, the scalability of these methods is a concern today due to the huge length of real-life traces and the slowness of interpolating provers.[16]

Using Passing Tests

Another technique, called "Angelic Debugging"[5] first proposed in 2011, explores the relationship between fault localization and fix localization rather closely, following the philosophy of defining a possible fault in terms of how it is fixed. In it, we explore the set of potential repairs that will make an observable error disappear. Since the landscape of syntactic repairs to try out is so vast, the technique finds, via symbolic execution and constraint solving, a value that makes the failing tests pass while continuing to pass the passing tests. Crucially, the technique utilizes the information contained in the passing tests to help identify fix locations. The technique proceeds in two steps. In the first, it attempts to find all the expressions in the program that are candidates for a fix; that is, a change made in that expression can possibly fix the program. The second step rules out those fix loca-

Table 2. Interpolant computation at each statement; the statements marked with • have the property that they do not alter the interpolant.

Statement	Interpolant after statement
$arg[1] = A$	$arg[1] = A$
$arg[2] = INET$ •	$arg[1] = A$
$arg[3] = NULL$ •	$arg[1] = A$
$hw_set_0 = 0$ •	$arg[1] = A$
$hw_0 = NULL$	$arg[1] = A \wedge hw_0 = NULL$
$ap_0 = NULL$ •	$arg[1] = A \wedge hw_0 = NULL$
$i_1 = arg[1]$	$i_1 = A \wedge hw_0 = NULL$
$guard_3 = (i_1 == A)$	$guard_3 = true \wedge hw_0 = NULL$
$ap_4 = arg[2]$ •	$guard_3 = true \wedge hw_0 = NULL$
$hw_set_5 = 1$	$guard_3 = true \wedge hw_0 = NULL \wedge hw_set_5 = 1$
$ap_{11} = guard_3 \, ? \, ap_4 : ap_0$ •	$guard_3 = true \wedge hw_0 = NULL \wedge hw_set_5 = 1$
$hw_set_{11} = guard_3 \, ? \, hw_set_5 : hw_set_0$	$hw_0 = NULL \wedge hw_set_{11} = 1$
$i'_1 = arg[3]$ •	$hw_0 = NULL \wedge hw_set_{11} = 1$
$guard_{13} = (hw_set_{11} == 0)$	$hw_0 = NULL \wedge guard_{13} = false$
$hw_{14} = DEFAULT$ •	$hw_0 = NULL \wedge guard_{13} = false$
$hw_{15} = guard_{13} \, ? \, hw_{14} : hw_0$	$hw_{15} = NULL$
$hw_{15} \neq NULL$	$false$

tions for which changing the expression would make previously passing tests fail. It does so *without* knowing any proposed candidate fix, again because the landscape of syntactic fixes is so vast; rather, it works on just the basis of a candidate fix "location."

Consider again the failing execution of `Test 2` on the program in Figure 1. We illustrate how the technique works, focusing first on statement 5. The technique conceptually replaces the right-hand-side expression by a "hole" denoted by *!!*, meaning an as-yet-unknown expression

```
5: hw_set = !!
```

The interpretation of *!!* is that an angel would supply a suitable value for it if it is possible to make the execution go to completion, hence the name of the technique. The angel is simulated by a symbolic execution tree (see Figure 4 for details on how to compute a symbolic execution tree), looking for a path along which the path formula is satisfiable. In our running example, the symbolic execution comes up with the following environment when going through the true branch at line 13 and expecting a successful termination

$e_{13T} = [hw_0 = NULL, i = A, ap = INET, hw_set = \alpha$
$guard_{13} = true, hw_{14} = DEFAULT$
$hw_{15} = DEFAULT;$
$\alpha = 0 \wedge hw_{15} \neq NULL \wedge hw_{15} == DEFAULT]$

and the following when going through the false branch

$e_{13F} = [hw_0 = NULL, i = A, ap = INET, hw_set = \alpha$
$guard_{13} = false, hw_{15} = NULL;$
$\alpha \neq 0 \wedge hw_{15} \neq NULL \wedge hw_{15} == DEFAULT]$

α represents the angelic value assigned at line 5. Recall we carried out concrete execution up to statement 5, with the same input as shown in formula ϕ earlier.

e_{13T} has a satisfiable condition when α is 0, whereas e_{13F} is not satisfiable due to the conflict on the value of hw_{15}. The execution can thus be correct in case the *guard* at line 13 evaluates to true.

Focusing next on statement 13, we find making the condition itself angelic, we can find a plausible successful execution of the program

```
13: if (!!) {
```

We omit the formulae, but they are similar to ones shown earlier in this article.

We now show the second step of the method. It rules out those fix locations for which changing the expression would make previously passing tests fail. Given a candidate fix location, it asks the following question for each of the passing inputs: Considering the proposed fix location a hole (*!!*), is there a way for the angel to provide a value for the hole that is *different* from the value observed at that location in normal execution on that input? If it is possible, then the fix location is a plausible fix location. The technique provides the angelic values by using symbolic execution.

First we consider the fix location of line 5, right-hand side. For `Test 1`, the hole in the fix location (line 5 of Figure 1) will be replaced by α, and $\alpha \neq 1$ added to the constraint, to represent difference from the value observed here in normal execution. More formally, the symbolic environment at line 5 will be

$e_5 = [hw_0 = NULL, i = A, ap = INET, hw_set = \alpha; \alpha \neq 1]$

From here on, symbolic execution will find a path that succeeds. Likewise, for `Test 2` and `Test 3`. The passing tests thus accept the proposed fix location as a plausible one.

Now consider the fix location of line 13, where we want the branch to have a different outcome. For `Test 1`, the environment at line 13 will be

$e_{13} = [hw_8 = ETHER, hw_set = 1, \ldots, guard_{13} = \alpha; \alpha \neq \text{false}]$

There is no successful execution given this environment. `Test 1` therefore rules out line 13 as a plausible fix location, deeming no syntactic variation of the condition is likely to fix the program.

Although the technique determines plausible fix locations and not fixes themselves, going from a fix location to a fix is not straightforward. Consider a candidate syntactic fix a human could provide for line 5. For example, using the

The scalability of symbolic analysis-based debugging methods crucially depends on the scalability of SMT constraint solving.

following fix at line 5 works for tests 1–3

```
5: hw_set = 0
```

The astute reader will notice this particular fix is not an ideal fix. Given another test

```
Test 4:  arp H ETHER A INET
         Expected output: ETHER
         Actual output: DEFAULT
```

This test will fail with the proposed fix, even though the location of the fix is the correct one. The correct fix is to eliminate the effect of line 5 altogether

```
5: hw_set = hw_set; // or delete the
statement
```

The example reflects the limitations in attempting to fix a program when working with an incomplete notion of "specification."

Using Other Implementations

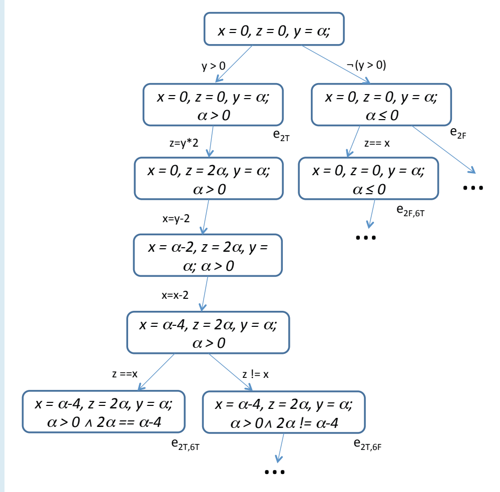
Programs are usually not written from scratch; rather, versions of a program are gradually checked in. When we introduce changes into a previously working version (where we say the version is "working," since all test cases pass), certain passing tests may thus fail. To debug the cause of the observed failure in such a failed test t, we can treat the behavior of t in the previous working version as a description of the programmer's intention.

The technique presented in Qi et al.,[22] called "Darwin," developed in 2009, executes the failing test t in the previous program version P, as well as the current program version P'. It then calculates the path conditions of t in both program versions along the execution trace of t in both program versions; see Figure 5 for explanation of how such path conditions are computed. Calculating path conditions leads to path conditions f and f'. One can then solve the formula $f \wedge \neg f'$ to find test input t' that follows the same path as t in the previous program version and a different path in the current program version. The execution of t' in the current program version P' can then be compared with the execution of the failing test t in current version P' in terms of differences in their control flow. That is, the behavior of t' in cur-

Figure 4. Symbolic execution tree.

Consider again the code fragment in Figure 2. Suppose input y is a symbolic input, with an unknown value of, say, α. In symbolic execution,[14] the store maps variables that may be concrete values or symbolic expressions. At an assignment, the store is updated with the evaluation of right-hand-side expression, which may be a symbolic expression. At a branch, if the decision involves a symbolic expression, both sides of the branch are executed in separate "threads," with corresponding branch conditions taken into account.

At line 2, two threads of symbolic execution will be created. In the first, the environment e_{2T} is $[x = 0, z = 0, y = \alpha; \alpha > 0]$, and the other, e_{2F}, is $[x = 0, z = 0, y = \alpha; \alpha \le 0]$; note we included the conditions encountered on the path thus far in the environment. These conditions appear following the semicolon (see also Figure 5.). Here is the symbolic execution tree for Figure 2



At line 6, e_{2T} forks into: $e_{2T,6T}$ and $e_{2T,6F}$. $e_{2T,6T}$ will be $[x = \alpha - 4, z = \alpha * 2, y = \alpha; \alpha > 0 \wedge \alpha - 4 = \alpha * 2]$, which will be discarded since the condition is unsatisfiable.

Symbolic execution tree construction is similar to the program formula construction in Figure 2. For this reason, it is also called "static symbolic execution." The difference is, in program formula, threads were merged with control-flow join points, whereas in symbolic execution tree, there is no merging.

Figure 5. Illustration of path conditions.

Consider yet again the program in Figure 2. Suppose we want to find the path condition of the only way to reach the error statement, or the path $\langle 1, 2, 3, 4, 5, 6, 8, 9\rangle$. We traverse forward along the sequence of statements in the given path, starting with a null formula and gradually build it up. All variables start with symbolic values. At any point during the traversal of the trace, we maintain an assignment store and a logical formula. The result is

- For every assignment, we update the symbolic assignment store; and
- For every branch, we conjoin the branch condition—or its converse if the branch is not taken—with the path condition; while doing so, we use the symbolic assignment store for every variable appearing in the branch condition.

At the end of the path, the logical formula captures the path condition. For the example path $\langle 1, 2, 3, 4, 5, 6, 8, 9\rangle$ in the given program, the path condition can be calculated as shown in the table here. Whenever the input satisfies $y > 0 \wedge 2y \neq y - 4 \wedge 2y > y - 4$, the program execution will trace exactly this path.

	Assignment store	Logical Formula
1	`{}`	$true$
2	`{}`	$y > 0$
3	`{(z,2y)}`	$y > 0$
4	`{(z,2y),(x, y-2)}`	$y > 0$
5	`{(z,2y),(x,y-2-2)}`	$y > 0$
6	`{(z,2y),(x,y-2-2)}`	$y > 0 \wedge 2y \neq y - 4$
8	`{(z,2y),(x,y-2-2)}`	$y > 0 \wedge 2y \neq y - 4 \wedge 2y > y - 4$
9	`{(z,2y),(x,y-2-2)}`	$y > 0 \wedge 2y \neq y - 4 \wedge 2y > y - 4$

Note this form of symbolic execution is similar to the one in Figure 4; the difference is one path is already given here, so there is no execution tree to be explored. This is sometimes also called "dynamic symbolic execution."

rent program P' is taken as the "specification" against which the behavior of the failing test t is compared for difference in control flow.

Such methods are based on semantic analysis, rather than a completely syntactic analysis of differences across program versions (such as running a `diff` between program versions). Being based on semantic analysis these debugging methods can analyze two versions with substantially different implementations and locate causes of error.

To illustrate this point, consider the fixed Address Resolution Protocol (ARP) implementation—Figure 1 with line 5 deleted—we discussed earlier as the reference version. This program will pass the test `Test 2`. Now assume a buggy program implementation with a substantially different programming style but with intention to accomplish the same ARP (see Figure 6). The test `Test 2` fails in this implementation

```
Test 2:
     arp A INET (failing test).
     Expected output: DEFAULT
     Observed output: INET
```

First of all, a simple `diff` of the program versions cannot help since almost the entire program will appear in the `diff`. A careful comparison of the two implementations shows the logic of the protocol the programmer would want to implement has been mangled in this implementation. The computation of `get hwtype(DEFAULT)` has been (correctly) moved. However, the computation of `get _ hwtype(optarg)`[a] is executed under an incorrect condition, leading to the failure in the test execution. A redundant check `ap != NULL` has slipped into line 8.

We now step through the localization of the error. For the test `arp A INET` the path condition in the reference version is (since i is set to arg[1]) as follows

$$f \equiv arg[1] = A$$

The path condition in the buggy implementation is as follows (since i is set to arg[1] and `ap` is set to arg[2] (via `optarg`)

$$f' \equiv arg[1]{=}A \wedge (arg[2]{\neq}NULL \vee arg[1]{=}H)$$

The negation of f' is the following disjunction

$$\neg f' \equiv arg[1]{\neq}A \vee \neg(arg[2]{\neq}NULL \vee arg[1]{=}H)$$

$f \wedge \neg f'$ thus has two possibilities to consider, one for each disjunct in $\neg f'$

1. $arg[1] = A \quad \wedge \quad arg[1] \neq A$
2. $arg[1] = A \quad \wedge \quad \neg(arg[2] \neq NULL \vee arg[1] = H)$

The first formula is not satisfiable, and the second one simplifies to

$$arg[1] = A \wedge arg[2] = NULL \wedge arg[1] \neq H$$

A satisfying assignment to this second formula is an input that shows the essential control-flow path deviation—in the defective program—from the failure-inducing input. This formula is satisfiable because arg[2] = $NULL$ is satisfiable, pointing to the condition being misplaced in the code. This is indeed where the bug lurks. Even though the entire coding style and the control flow in the buggy implementation was quite different from the reference implementation, the debugging method is thus able to ignore the differences in coding style in the buggy implementation. Note arg[2] = $NULL$ is contributed by the branch condition `ap` != NULL, a correlation an automated debugging method can keep track of. The method thus naturally zooms into the misplaced check `ap` != NULL through a computation of satisfiability of the deviations of the failing test's path condition.

One may question the choice of considering the past version as a specification of what the program is supposed to achieve, a question that arises because the software requirements of an earlier program version may differ from the requirements of the current program version. However, the comparison works as long as the program features are common to both versions and the requirements are unchanged.

Perspectives

We have explored symbolic execution and constraint solving for software debugging. We conclude by discussing scalability and applicability of the presented techniques, along with possible future research directions.

Scalability and applicability. The scalability of the techniques described here need greater investigation. Due to the semantic nature of the analysis, symbolic execution-based debugging of regression errors[22] has been adapted to multiple settings of regression, resulting in wide applicability, including regressions in a program version as opposed to previous version; regression in an embedded software (such as Linux Busybox) as opposed to reference software (such as GNU Coreutils);[3] and regression in a protocol implementation (such as miniweb Web server implementing the http protocol) as opposed to a reference implementation of the protocol, as in the Apache Web server. In these large-scale applications, the symbolic analysis-based regres-

a Assume get `hwtype(INET)` returns `INET`.

Figure 6. Assume line 5 in Figure 1 was removed yielding the correct program for the code fragment; here we show a different implementation of the same code.

```
 1  hw_set = 0; hw = NULL;
 2  while (i = getopt(....)) // this sets optarg
 3  {
 4      if (i == 'A') {
 5              ap = getaftype(optarg);
 6              hw = get_hwtype(DEFAULT);
 7      }
 8     if (ap != NULL || i == 'H') {
 9              hw = get_hwtype(optarg);
10              hw_set = 1;
11      }
12 }
13 assert(hw  != NULL);
14 print hw;
 ...
```

sion debugging methods of Banerjee et al.[3] and Qi et al.[22] localized the error to within 10 lines of code in fewer than 10 minutes.

Among the techniques covered here, the regression-debugging methods[3,22] have shown the greatest scalability, with the other techniques being employed on small-to-moderate-scale programs. Moreover, the scalability of symbolic analysis-based debugging methods crucially depends on the scalability of SMT constraint solving.[8] Compared to statistical fault-localization techniques, which are easily implemented, symbolic execution-based debugging methods still involve more implementation effort, as well as greater execution time overheads. While we see much promise due to the growth in SMT solver technology, as partly evidenced by the scalability of the regression-debugging methods, more research is needed in symbolic analysis and SMT constraint solving to enhance the scalability and applicability of these methods.

Note for all of the presented debugging methods, professional programmers need user studies to measure programmer productivity gain that might be realized through these methods. Parnin and Orso[21] highlighted the importance of user studies in evaluating debugging methods. The need for user studies may be even more acute for methods like BugAssist that provide an iterative exploration of the possible error causes, instead of providing a final set of diagnostic information capturing the lines likely to be the causes of errors. Finally, for the interpolant-based debugging method, the issue of choosing suitable interpolants needs further study, a topic being investigated today (such as by Albarghouthi and McMillan[2]).

Other directions. Related to our topic of using symbolic execution for software debugging, we wish to say symbolic execution can also be useful for "bug reproduction," as shown by Jin and Orso.[11] The bug-reproduction problem is different from both test generation and test explanation. Here, some hints may be reported about the failing execution by on-the-field users in the form of a crash report, and these hints can be combined through symbolic execution to construct a failing execution trace.

Finally, the software-engineering community has shown much interest in building semiautomated methods for program repair. It would be interesting to see how symbolic execution-based debugging methods can help develop program-repair techniques. The research community is already witnessing development of novel program-repair methods based on symbolic execution and program synthesis.[17,18,20]

Acknowledgments

We would like to acknowledge many discussions with our co-authors in Banerjee et al.,[3] Chandra et al.,[5] and Qi et al.[22] We also acknowledge discussions with researchers in a Dagstuhl seminar on fault localization (February 2013) and a Dagstuhl seminar on symbolic execution and constraint solving (October 2014). We further acknowledge a grant from the National Research Foundation, Prime Minister's Office, Singapore, under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

References

1. Agrawal, H. and Horgan, J. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (White Plains, NY, June 20–22). ACM Press, New York, 1990.
2. Albarghouthi, A. and McMillan, K.L. Beautiful interpolants. In *Proceedings of the 25th International Conference on Computer-Aided Verification, Lecture Notes in Computer Science 8044* (Saint Petersburg, Russia, July 13–19). Springer, 2013.
3. Banerjee, A., Roychoudhury, A., Harlie, J.A., and Liang, Z. Golden implementation-driven software debugging. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering* (Santa Fe, NM, Nov. 7–11). ACM Press, New York, 2010, 177–186.
4. Cadar, C. and Sen, K. Symbolic execution for software testing: Three decades later. *Commun. ACM 56*, 1 (Jan. 2013), 82–90.
5. Chandra, S., Torlak, E., Barman, S., and Bodik, R. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering* (Honolulu, HI, May 21–28). ACM Press, New York, 2011, 121–130.
6. Christ, J., Ermis, E., Schaff, M., and Wies, T. Flow sensitive fault localization. In *Proceedings of the 14th International Conference on Verification Model Checking and Abstract Interpretation* (Rome, Italy, Jan. 20–22). Springer, 2013.
7. Cleve, H. and Zeller, A. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, May 15–21). ACM Press, New York, 2005.
8. de Moura, L. and Björner, N. Satisfiability modulo theories: Introduction and applications. *Commun. ACM 54*, 9 (Sept. 2011), 69–77.
9. Ermis, E., Schaff, M., and Wies, T. Error invariants. In *Proceedings of the 18th International Symposium on Formal Methods, Lecture Notes in Computer Science* (Paris, France, Aug. 27–31). Springer, 2012.
10. GNU Core Utilities; http://www.gnu.org/software/coreutils/coreutils.html
11. Jin, W. and Orso, A. BugRedux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering* (Zürich, Switzerland, June 2–9). IEEE, 2012.
12. Jones, J.A., Harrold, M.J., and Stasko, J.T. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, FL, May 19–25). ACM Press, New York, 2002.
13. Jose, M. and Majumdar, R. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32nd International Conference on Programming Language Design and Implementation* (San Jose, CA, June 4–8). ACM Press, New York, 2011, 437–446.
14. King, J.C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (July 1976) 385–394.
15. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., and Jordan, M.I. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, June 12–15). ACM Press, New York, 2005, 15–26.
16. McMillan, K.L. An interpolating theorem prover. *Theoretical Computer Science 345*, 1 (Nov. 2005), 101–121.
17. Mechtaev, S., Yi, J., and Roychoudhury, A. DirectFix: Looking for simple program repairs. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering* (Firenze, Italy, May 16–24). IEEE, 2015, 448–458.
18. Mechtaev, S., Yi, J., and Roychoudhury, A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, TX, May 14–22) ACM Press, New York, 2016.
19. Morgado, A., Heras, F., Liffiton, M., Planes, J., and Marques-Silva, J. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints 18*, 4 (2013), 478–534.
20. Nguyen, H.D.T., Qi, D., Roychoudhury, A., and Chandra, S. SemFix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering* (San Francisco, CA, May 18–26). IEEE/ACM, 2013, 772–781.
21. Parnin, C. and Orso, A. Are automated debugging techniques actually helping programmers? In *Proceedings of the 20th International Symposium on Software Testing and Analysis* (Toronto, ON, Canada, July 17–21) ACM Press, New York, 2011, 199–209.
22. Qi, D., Roychoudhury, A., Liang, Z., and Vaswani, K. DARWIN: An approach for debugging evolving programs. *ACM Transactions on Software Engineering and Methodology 21*, 3 (2012).
23. Weiser, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.
24. Zeller, A. Yesterday my program worked. Today it fails. Why? In *Proceedings of the Seventh Joint Meeting of European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering, Lecture Notes in Computer Science* (Toulouse, France, Sept. 1999). Springer, 1999, 253–267.
25. Zeller, A. *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier, 2006.

Abhik Roychoudhury (abhik@comp.nus.edu.sg) is a professor of computer science in the School of Computing at the National University of Singapore and an ACM Distinguished Speaker and leads the TSUNAMI center, a software-security research effort funded by the Singapore National Research Foundation.

Satish Chandra (schandra@acm.org) leads the advanced programming tools research team at Samsung Research America, Mountain View, CA, and is an ACM Distinguished Scientist.

© 2016 ACM 0001-0782/16/07 $15.00