Angelic Debugging

Satish Chandra
IBM Research
satishchandra@us.ibm.com

Emina Torlak
IBM Research
etorlak@us.ibm.com

Shaon Barman
UC Berkeley
sbarman@cs.berkeley.edu

Rastislav Bodik
UC Berkeley
bodik@cs.berkeley.edu

ABSTRACT

Software ships with known bugs because it is expensive to pinpoint and fix the bug exposed by a failing test. To reduce the cost of bug identification, we locate expressions that are likely causes of bugs and thus candidates for repair. Our symbolic method approximates an ideal approach to fixing bugs mechanically, which is to search the space of all edits to the program for one that repairs the failing test without breaking any passing test. We approximate the expensive ideal of exploring syntactic edits by instead computing the set of values whose substitution for the expression corrects the execution. We observe that an expression is a repair candidate if it can be replaced with a value that fixes a failing test and in each passing test, its value can be changed to another value without breaking the test. The latter condition makes the expression *flexible* in that it permits multiple values. The key observation is that the repair of a flexible expression is less likely to break a passing test. The method is called *angelic debugging* because the values are computed by angelically nondeterministic statements. We implemented the method on top of the Java PathFinder model checker. Our experiments with this technique show promise of its applicability in speeding up program debugging.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Testing, Debugging, Validation

Keywords

Angelic non-determinism, debugging, symbolic execution, tests

1. INTRODUCTION

Software producers often ship code with known faults that are deemed too expensive to fix and unlikely to occur in practice. Isolating causes of these faults from failing tests is a labor-intensive process, with considerable costs. Given that compute cycles are abundant and inexpensive, judicious use of automation for fault isolation could greatly improve software quality, as well as reduce costs.

In this paper, we propose a computer-*assisted* debugging methodology, in which value-based reasoning is used to identify plausible *repair candidates* in a faulty program. We consider only those faulty programs in which the defect can be repaired by altering at most one expression. A repair candidate is an expression that can be modified to correct the failing test(s) without breaking any passing tests. In principle, we could identify repair candidates by exploring (a subset of) valid syntactic variations of all expressions in the program. This, however, tends to be too computationally demanding in practice.

Our idea is to approximate syntactic changes with value changes. Instead of checking if the program can be fixed by replacing an expression e with some expression e', we check whether a failing test can be repaired by replacing the value v of e with some value v'. If v' fixes the test, we assume it is possible to find an expression e' that evaluates to v'. If no such v' exists, we can filter out e from the set of repair candidates.

Of course, the value-replacement check based only on a failing test may not be able to filter out repair candidates that could break some of the passing tests. Our approach takes into account the effects of repairing e in *passing* tests as follows: if e evaluates to w in a passing test, but the same test continues to pass if w is replaced with a value $w' \neq w$, then we have a level of confidence that e can be replaced with a syntactically different expression e' (as we shall see later, this is an approximate test).

In summary, if replacing an expression e with a value v fixes the failing test, then repairing e has the potential to fix the failing test. If replacing e with a value v different than e computes does not break a passing test, then we have some freedom to change e; we say that it is *flexible* and a good repair candidate.

Our methodology is called *angelic* debugging and comprises the following steps:

1. The programmer—based on intuition, observation of runtime logs, or feedback from fault localization tools (e.g., Tarantula [14])—demarcates a scope in the code where he believes the defect might lie. By selecting a scope, the programmer expresses the hypothesis that altering some expression within the scope can resolve the bug.

2. The computer tries to validate this hypothesis. It automatically examines all expressions in the scope to find plausible repair candidates. For each such candidate, the computer demonstrates the existence of one alternative value that would have rescued the failing test case.

3. The computer then factors in information from any passing test cases to automatically eliminate inflexible repair candi-

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

```
1  public int classify(int a, int b, int c) {
2    int retval;
3    if (a >= b && b >= c) {
4      if (a == c || b == c) {
5        if (a == b && a == c)
6          retval = EQUILATERAL;
7        else
8          retval = ISOSCELES;
9      } else {
10       if (a*a != b*b + c*c) {
11         if (a*a < b*b + c*c) {
12           retval = ACUTE;
13         } else {
14           retval = OBTUSE;
15         }
16       } else {
17         retval = RIGHT;
18       }
19     }
20   } else {
21     retval = ILLEGAL;
22   }
23   return retval;
24 }
```

Figure 1: Triangles example from DeMillo [8].

dates. A repair candidate is inflexible if changing the candidate expression can rescue a particular failing test, but any change would break one of the previously passing tests. If all repair candidates are eliminated in this step, then the programmer has either not chosen the scope correctly, or there is indeed no fix to the program that can pass all test cases.

4. The programmer attempts to devise a suitable replacement expression for each of the remaining repair candidates, one at a time, until one works. A successful replacement expression must give passing results on the hitherto failing input, as well as on the previously passing inputs. The search for such a replacement expression is not computer assisted, as the space of possible syntactic expressions is unbounded. However, the programmer can draw inspiration from the sample alternate value demonstrated by the second step.

Example. Consider the code in Figure 1. It is a slightly modified version of an example from the paper by DeMillo et al. [8] on mutation testing. The program takes as input the lengths of three sides of a triangle, presented in non-decreasing order, and it is supposed to classify the triangle as one of equilateral, isosceles, right angled, acute, or obtuse.

Table 1 shows eight sample inputs to the program, also taken from the DeMillo paper. The code works correctly for inputs T1-T7, but not for T8.

Suppose the program is known to be fixable, in the specific sense that there is a single expression that, if altered, will let all of the test inputs pass. It is the task of the programmer to find the faulty expression and repair it.

In the first step, the programmer has to hypothesize a scope. We assume that the entire program is in scope. In the second step, the computer figures out the repair candidates for the failing test T8. These are shown in Table 2. Each repair candidate could have salvaged this *particular* execution by evaluating to a different value than it did. The column titled "alt value" lists the alternate value that would have made T8 pass. For example, the first row in the table says that if the value 7 were to be produced in place of the highlighted expression a, whose runtime value is 26, T8 would pass.

For brevity, we have considered only reads of local variables and constants, but there is no conceptual reason to exclude arithmetic and logical expressions (e.g., the result of a == c). Among the kinds of expressions considered, these are the only possible repair candidates.

The third step uses the passing test cases to filter out the inflexible repair candidates, leaving the right choice and one wrong choice. This is shown in the last column of Table 2. For example, the last row says it is no good changing the value of ACUTE at line 12 to ISOSCELES; doing so will rescue T8, but break test T6, whose *only* correct outcome is in fact ACUTE. This much would be obvious to a programmer. The other two choices that our technique eliminates from consideration on the basis of test T3 may not be immediately obvious.

Given the information in Table 2, a programmer would consider each of the remaining two candidates in turn, and try to find a *syntactic replacement* that produces an acceptable value for T8, and that works for all the other test cases (T1-T7) as well. (Note that the concrete values that our technique suggests are specific to T8.) Of these two candidates, the first choice is in fact futile, but we cannot get that information from the passing tests. The second choice is the real bug: it should be b in place of c.

We call our methodology angelic debugging, because it relies on *angelic* nondeterminism. Imagine that an expression is replaced with a call to an *oracle* that returns a value, chosen nondeterministically. The oracle is both omniscient and co-operative, in the sense that if a suitable value exists—one that makes the failing input now pass—the oracle is guaranteed to return it. While supporting angelic nondeterminism is computationally expensive, it is becoming increasingly viable for selective use, thanks to recent advancements in decision procedures. In our case, angelic non-determinism is used only on one expression at a time. This limits the kinds of defects on which our technique is effective, but enhances computational feasibility. Nondeteminism has long been used in testing to create an adversary (a demonic oracle), but its use as an ally is less common.

A key novelty in this work is in how we use information from passing tests to filter out repair candidates that will not work out. While the programmer still needs to go through the laborious task of finding replacement expressions for multiple repair candidates, he does not need to carry out the Sisyphean effort of looking for replacement expressions for repair candidates that are "doomed" by another test case. Without first positing a replacement expression, he would not know whether it would work out or not, and the space of replacement expressions can be huge.

Previous work on fault localization that has leveraged information from a suite of passing tests has relied on indirect information. The basic idea is that a failing run has an anomalous behavior in

Test	a	b	c	Expected outcome
T1	2	12	27	ILLEGAL
T2	5	4	3	RIGHT
T3	26	14	14	ISOSCELES
T4	19	19	19	EQUILATERAL
T5	9	6	4	OBTUSE
T6	24	23	21	ACUTE
T7	7	5	6	ILLEGAL
T8	26	26	7	ISOSCELES actual outcome: ACUTE

Table 1: Inputs for Fig. 1

line#	code	alt value (T8)	inhibitor
4	if (**a** == c || b == c) {	7	
4	if (a == **c** || b == c) {	26	
4	if (a == c || **b** == c) {	7	T3
4	if (a == c || b == **c**) {	26	T3
12	retval = **ACUTE**	ISOSCELES	T6

Table 2: Repair candidates for Fig. 1.

some regard, when compared to passing runs. The anomaly in behavior could be executing statements that are rarely, if ever, executed by passing tests; or it could be violating invariants that have been discovered to hold during passing executions. This information, while useful in fault localization, does not directly answer the question "Can I change a specific expression?" We believe that for debugging applications, this is the more important question.

We have built a tool, ANGELINA, that supports angelic debugging. It has been built on top of the JAVA PATHFINDER model checker [15]. ANGELINA accepts as input a Java program, a failing test case, and a suite of passing test cases. The default scope is the entire program, but a programmer can specify scopes at various granularities: a class file, a method, or a range of lines. ANGELINA outputs information of the kind shown in Table 2. ANGELINA takes no specification or annotation; it bases its findings only on the input mentioned above.

Remarkably, ANGELINA not only finds faulty arithmetic and logical expressions, but also faulty heap references for which it suggests alternative heap values: this feature is crucial for handling typical bugs in Java programs.

We have used ANGELINA on several small examples, and one large Java application taken from the SIR repository [24].

Contributions. This paper makes the following contributions:

1. We propose the angelic debugging methodology, in which we take advantage of plentiful compute cycles—instead of the programmer's time—to answer questions about where to fix a bug in a program.
2. We present a novel way of leveraging the information in passing test inputs to eliminate non-productive repair candidates.
3. We have implemented the automated parts of this methodology in a tool, and have evaluated the tool on a number of small and one large Java application.

Outline. The rest of the paper is organized as follows. Section 2 presents technical details of computing a filtered list of repair candidates. Section 3 describes how angelic execution is implemented in ANGELINA. Section 4 presents an evaluation of the tool, and Section 5 presents related work.

2. OVERALL APPROACH

In this section, we present a technical overview of angelic debugging. For expository reasons, we assume that each expression has a single dynamic occurrence; later sections will clarify how we deal with loops and procedure calls in practice. Hereafter, when we refer to an expression e, we mean a specific occurrence of e in the program.

Our approach uses *angelic* execution of a program. In angelic execution, a selected expression in the program is interpreted as a

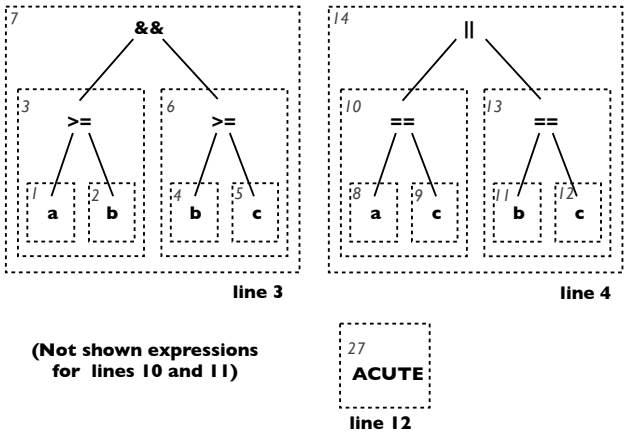

Figure 2: Portion of the AST from Figure 1

query to an angel, whose job is to return a value—if one exists—that would make the program pass. We denote $P[\alpha/e]$ as the program obtained by replacing the occurrence of e in P by a symbolic variable α. For a given expression e and input I, $AngelicTest$ is defined as follows:

$$AngelicTest(P, I, e) = \exists \alpha. Test(P[\alpha/e], I)$$

The expression $Test(P, I)$ above is true if interpreting P on I produces the expected outcome (provided with I) within a bounded amount of time. It is false otherwise.

Recall that a repair candidate is an expression that, if replaced by another expression, can rescue a failing run. An initial set of repair candidates can be specified as follows:

$$\{e \mid AngelicTest(P, I_f, e)\}$$

To obtain an initial list of repair candidates, our technique carries out angelic execution separately for each expression evaluated on the path taken by the failing execution triggered by input I_f. (Of course, once an angelic value is obtained, the program may proceed along a different, successful path if one exists.) For compound expressions, the subexpressions are tested for being repair candidates as well. Figure 2 shows the various expressions that are on the path executed by the failing input for the program in Figure 1. For each e_i, where i ranges from 1 to 27, we invoke $AngelicTest(P, I_f, e_i)$ to compute the set of repair candidates. This process identifies the five repair candidates mentioned in Table 2, as the other expressions cannot rescue the failing run no matter what value they return.

Since each expression on the faulty path is considered for angelic valuation, all potential repair candidates will indeed be found. (Various static analyses can be applied to optimize this process.) Applying $AngelTest$ will also yield a concrete value for each repair candidate that would rescue the given test input.

Next, we leverage the passing test inputs, if any. Given a repair candidate e, and a passing test input I_p, could the program still pass if you substitute e with a *different* value than what e evaluates to on this input? If the answer is no, it is not a good repair candidate. This question can also be answered by angelic execution, via the *flexibility* check:

$$FlexTest(P, I, e) = \exists \alpha.(Test(P[\alpha/e], I) \wedge \alpha \neq Eval(P, I, e))$$

We define $Eval(P, I, e)$ as the value obtained by evaluating the expression e in the course of interpreting the program P on the

input I. If e is not visited while executing P on I, $Eval(P, I, e)$ is undefined and $FlexTest(P, I, e)$ is trivially true.

Given a failing input I_f and passing inputs $I_{p_1} \ldots I_{p_k}$, the good repair candidates are:

$$\{e \mid AngelicTest(P, I_f, e) \wedge \forall i \in [1..k].FlexTest(P, I_{p_i}, e)\}$$

For example, consider the last row of Table 2. The only passing test case in which line 12 is visited is T6, so it is pertinent to ask the question only for T6. T6 can pass only when the value of retval at line 12 is ACUTE; no other value would let T6 pass. Hence, T6 rules out that repair candidate. Line 12 sets the (constant) return value, which obviously is not the right fix and, the flexibility test discovers that it is not the right fix. In other cases, inflexibility might not be immediately obvious, e.g., the repair candidates ruled out by T3.

We note that there is no guarantee that for every "good" repair candidates that our technique produces, there exists a suitable replacement expression (i.e., one that would work out for the failing as well as all the passing tests.) The value of our technique is in eliminating from consideration those repair candidates for which no suitable replacement expression is likely to exist.

Rationale for *FlexTest*. If an expression e in the program is the right repair candidate, it must be fine to replace e with some as-yet-undetermined expression e'. The expression e is not the right repair candidate if replacing it with *any* expression e' will make one or more of the passing test cases fail. Can we take advantage of passing test cases *without* knowing e'?

The ideal flexibility question that we want to ask is this:

> (*Ideal*) Given a repair candidate—i.e., an expression e—and given a passing test case T, is it possible for T to continue to pass if we replace e by a different expression e'?

If the answer is no, then we do not consider e to be the place to fix, because it can break T (and we are allowed only one fix to the program.)

Since the ideal question is too expensive to answer, *FlexTest* carries out a more readily answerable, but approximate, check. It asks the following question on *values*:

> (*Approximate*) Given a repair candidate e, and given a passing test case T, suppose that e produces the value w when T is run. Is is possible for T to continue to pass if e is replaced by some w' where $w \neq w'$? If T does not encounter e, the check returns true by default.

From the perspective of a passing test, if the answer to the approximate question is yes, then the answer to the ideal question is also yes. On the other hand, if the answer to approximate question is no, then the answer to the ideal question is *probably* no; but it is not guaranteed to be negative. Consider the following example:

```
t = x + y
```

Suppose that e is y; e' is 1; and t must be x+1 later in the execution. Then, on a passing test case for which y happens to be exactly 1 at this point in the code, the approximate check would say it is inflexible. The ideal check, however, would have allowed e'.

We believe that in practice this is not a problem; indeed, we have not found this to be a problem in using an implementation of this idea on various examples. Since the error is "one sided", we can reduce the mathematical likelihood of being wrong—eliminating a repair candidate unnecessarily—by relying on many passing tests, together with making the notion of flexibility from a Boolean to a quantitative property that would estimate the likelihood that an expression is flexible.

Limitations. If the desired fix to a program is a single expression that needs to be altered—or any one of a set of single expressions—our technique will eventually find it. We call such programs *1-fixable*.

If a program is not 1-fixable, then our technique can suggest a repair candidate that is not the desired fix, but that can often inspire the actual fix. For example, if the program is missing a certain assignment to a field, then a fix might be suggested at a point downstream where the field is used (assuming there is just one such use.)

Our technique also inherits the limitations of constraint solvers used in the implementation of the angelic nondeterminism. It is well known that real programs can defeat the capabilities of current generation constraint solvers. Supplementing the solver with dynamic execution will sidestep these problems, potentially at the cost of reducing scalability.

3. ANGELIC EXECUTION

There are many ways to realize the angelic execution environment described in the previous section (e.g., [5]). Our tool, ANGELINA, employs a symbolic execution [18] engine that is based on an explicit-state model checker [15]. It implements the functions $AngelicTest$ and $FlexTest$ in three steps.

In the first step, we execute the failing test concretely, collecting the labels of all executed statements in a trace f. Each label in f gives a unique identity to a dynamically occurring bytecode instruction. It can be thought of as the bytecode index that an executed instruction would have if all loops in the program were unwound and all method calls inlined.

In the second step, the failing test is re-executed $|f|$ times to find all statements in f that are potential *repair candidates*. The i^{th} execution in this phase is concrete until it reaches the i^{th} statement $f[i]$, at which point we replace the concrete value returned by $f[i]$ with a symbolic value s_i. From then on, the values of all branch predicates that depend on the output of $f[i]$ are encoded as constraints over s_i. Section 3.1 describes this process in more detail. If there is a path through the program in which all branch constraints are satisfied, and which does not lead to a failure, then we have found a value that fixes the failing test and $f[i]$ is placed in the repair candidate set R.

The third step of the analysis uses the available passing tests to prune R. Each passing test is first executed concretely to collect a trace p. Next, for each statement $p[i]$ that is in R, we execute the prefix $p[0..i]$ of p concretely. This includes the statement $p[i]$, whose concrete execution produces the concrete value c_i. We then replace c_i with a symbolic value s_i and search for a failure-free path through the program in which s_i is different from c_i. If such a path is found, we say that $p[i]$ is *flexible* and keep it as a repair candidate. Otherwise, $p[i]$ is *inflexible*, and we remove it from R.

3.1 Symbolic Execution

Symbolic Execution of Integer Programs. To execute a program in which an integer-valued expression is treated symbolically, our tool employs the classic approach [18] of searching for a failure-free path in the program's *symbolic execution tree*. An example of such a tree is shown in Fig. 3. The nodes in the tree represent pro-

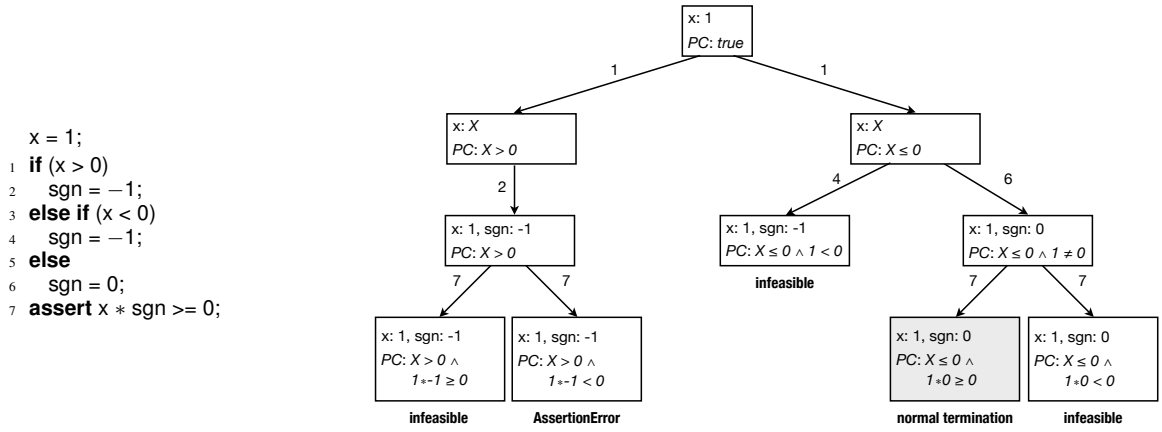

Figure 3: A symbolic execution tree for an integer program. The tree represents a symbolic execution of the sample program in which the concrete value of x on line 1 is replaced with the symbolic value X.

gram *states*, and the edges are transitions between states. Each state includes the (symbolic or concrete) values of program variables; a *path condition* (PC); and a program counter. The path condition is a boolean formula over symbolic values that encodes the conditions under which a given execution path can be taken; i.e., the path condition is satisfiable only along feasible paths. The program counter, shown as the edge label, specifies the next statement to be executed from the given state.

The symbolic execution tree on the right of Fig. 3 corresponds to the buggy implementation of the signum function shown on the left. The tree illustrates an execution of the program in which the concrete value of the variable x on line 1 is replaced with the symbolic value X. Note that all other occurrences of x remain concrete. The root of the tree represents the initial state of the execution, in which x has the concrete value 1 and the path condition is $true$. At each branch point, the PC is updated to reflect which path is taken. For example, evaluating the conditional on line 1 and taking the true-branch leads to the left child of the root, in which x has the symbolic value X and the path condition is $X > 0$. Whenever the path condition becomes false, or we reach an error state, the execution along that path terminates, and the search backtracks. In this example, ANGELINA explores three different paths (two infeasible and one erroneous) before reaching the highlighted passing state. The remaining infeasible path is shown for completeness; the search terminates as soon as the highlighted state is discovered.

Symbolic Execution of Heap-Manipulating Programs. To analyze a program in which a reference value is symbolic, we employ a variant of the generalized symbolic execution algorithm by Khurshid et al. [17]. The algorithm works like the basic symbolic execution described above, except when it encounters a field access to a symbolic reference s. In this case, we perform a *lazy concretization* of s and proceed as before. The lazy concretization process involves choosing a concrete value for s; recording that choice in the path condition; and, if the updated path condition is satisfiable, replacing all occurrences of s with the chosen concrete value. In our framework, the possible concrete values for s include null; an existing concrete reference of the same type as s ; and a reference to a new object (also of the same type as s) with all of its fields initialized to fresh symbolic values. In contrast to previous work [17, 21] where only certain aliasing relationships are considered, our choice of concrete values accounts for all aliasing that s may exhibit.

Figure 4 shows the generalized symbolic execution tree for the buggy heap-manipulating program displayed on the left. The tree characterizes the execution of the program in which the reference produced by the new expression on line 2 is treated symbolically. Each program state (i.e., tree node) is shown together with its corresponding heap, as indicated by the gray shading. An object in the heap is represented by a table that maps field names to values. Symbolic values are italicized, and concrete references are represented by arrows.

Executing the first two lines of the program results in the creation of concrete objects N1 and N2, whose fields are initialized to default values. To avoid cluttering the figure, we pretend that the execution of line 2 has no effect on the heap; it simply introduces a new symbolic reference $N2$ to which the variable n2 is bound. Line 3 sets the "next" field of N1 to the symbolic reference $N2$. Line 4 dereferences n2, which triggers the concretization of $N2$. Setting $N2$ to null leads to a null pointer exception, causing the search to backtrack. The next choice, N1, leads to an infeasible path and an assertion failure. (Note that the conditional on line 4 evaluates to true in this case.) The last choice—setting $N2$ to a fresh object N3 with symbolic field values—results in a failure-free path. The remaining erroneous paths are shown for completeness.

3.2 Implementation

ANGELINA is implemented on top of the JAVA PATHFINDER (JPF) model checker [15]. JPF is a Java Virtual Machine that is capable of backtracking to a given point in the execution and proceeding along a different path. Our implementation is patterned after the SYMBOLIC JPF (SJPF) tool [21]. Like SJPF, we use the extension mechanisms provided by JPF to force non-standard interpretation of instructions that use or define symbolic values. Path conditions are checked for satisfiability using the Choco constraint solver [6].

We avoid potentially infinite symbolic executions of loops and recursion by bounding the number of terms in the path condition. As soon as the path condition reaches a certain (user-determined) size, the corresponding path is abandoned. The current prototype also abandons paths on which a symbolic value is used to index into an array.

Procedure calls are handled automatically by JPF's dispatch mechanism. We simply ensure that calls to symbolic receivers can be resolved by keeping track of the runtime type of each symbolic reference.

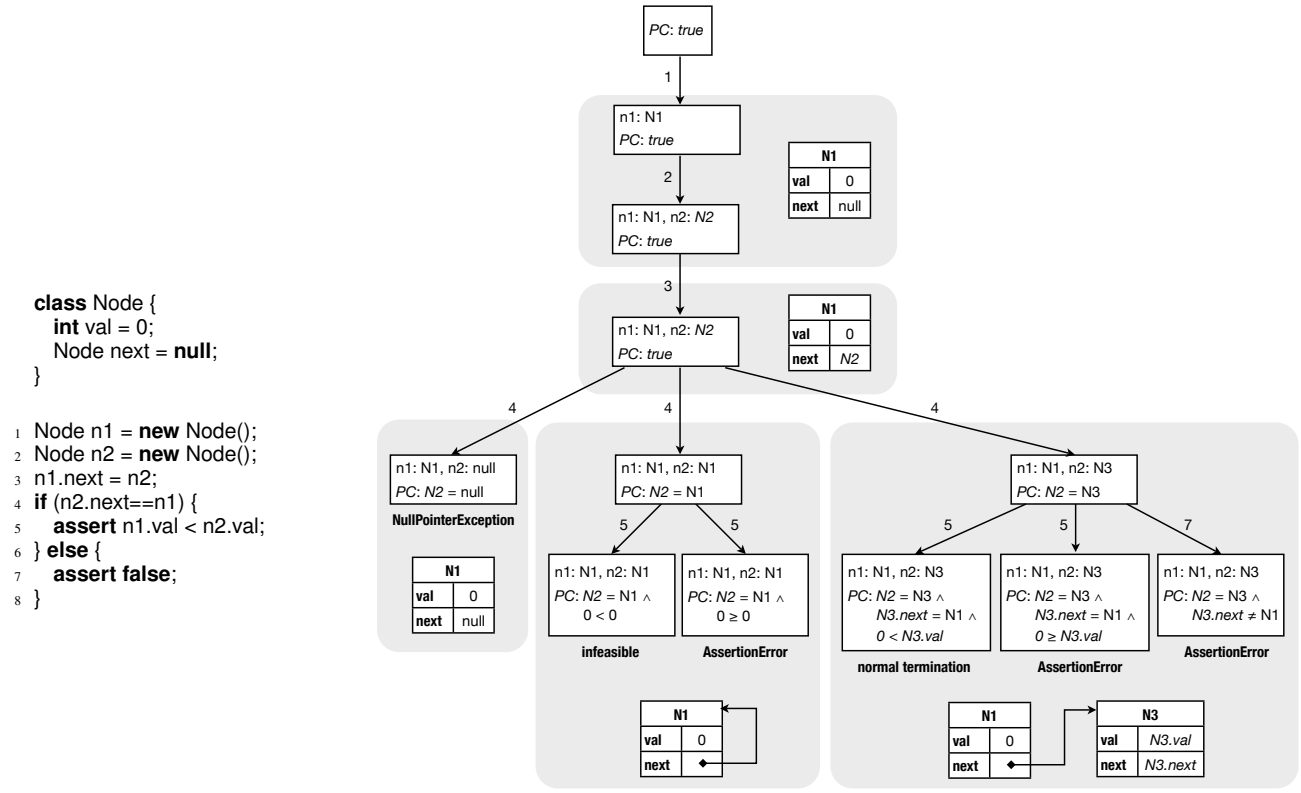

Figure 4: A symbolic execution tree for a heap-manipulating program. The tree represents a symbolic execution of the sample program in which the reference value produced by the new expression on line 2 is treated symbolically.

One subtle issue arises when $FlexTest$ (see Sec. 2) is performed on a reference. Since our solution to heap constraints allows manufacture of new objects, the requirement of dis-equality ($\neq$) between the $Eval$ value and the newly constructed value (α) is satisfied trivially. If a programmer's notion of equality is not based on reference equality, but rather on an equals predicate, then the flexibility check is not meaningful. To implement a meaningful equality test, we preserve a snapshot of the field values as observed in the default execution—i.e., as computed by $Eval$. In principle, we would then have JPF execute the programmer-provided equals method to compare the snapshot to the replacement value α. This is somewhat intricate to implement, however, so our current prototype carries out a shallow comparison of the top-level fields.

4. EVALUATION

To assess the feasibility and usefulness of angelic debugging, we applied ANGELINA to several programs with known defects. This section presents the results we obtained on two representative samples: a small integer program and a large heap-manipulating application.

4.1 Zunebug Example

Our integer example, shown in Figure 5, is taken from Weimer et al. [28]. The program takes as input a number of days as input, and returns the year in which that many days would elapse starting from beginning of 1980. (Table 3 shows samples of input and output of this program.) We have changed it slightly from Weimer's version to make it 1-fixable. The original buggy program required both a statement insertion and a statement deletion to fix it. We have

```
1  public static int zunebug(int days) {
2      int year = 1980;
3      while (days > 365) {
4        boolean leap = isLeapYear(year);
5        if (leap) {
6          if (days > 366) {
7            days −= 366;
8            year += 1;
9          } else {
10         }
11         days −= 366;
12       } else {
13         days −= 365;
14         year += 1;
15       }
16     }
17     return year;
18 }
```

Figure 5: Zunebug code.

carried out the insertion, so the only bug is a superfluous statement that needs to be deleted.

The program was tested on inputs shown in Table 3; T6 is the failing input. The repair candidates, and the inflexibilities found are reported in Table 4. The return statement was excluded from scope. Note that in this program, alternate values are one of potentially many valid possibilities. The code inside isLeapYear was excluded from scope.

Of the 7 repair candidates, 2 are ruled out as inflexible by passing tests. The real fix is at line 7: the line should be omitted, or

Test	days	Expected outcome
T1	1	1980
T2	365	1980
T3	366	1980
T4	367	1981
T5	731	1981
T6	732	1982
		actual outcome: 1981

Table 3: Tests for Zunebug.

line#	code	alt value (T6)	inhibitor
3 (2nd)	while (**days** > **365**) {	366, -1	
11	days -= **366**	0	
8	year += **1**	2	T4, T5
7	days -= **366**	0	
5	if (**leap**) {	false	T3
4	leap = isLeapYear(**year**)	1981	

Table 4: Repair candidates for Zunebug.

equivalently days should be decremented by 0. This fix causes the while loop to be executed two times, as required for the input value of 732. The repair candidates also show 4 alternate ways in which this can be achieved from the perspective of the failing run. Specifically, one option is to get the loop condition to pass one more time as shown by the first line in the table. Although it might seem as if the loop condition—the second instance of it—is flexible, in fact its operands are the ones that are flexible, not the condition itself. That means that for each passing run, either of those operands could have had at least one other value without impacting the outcome of the run; this is easy to happen with inequalities.

4.2 JTopas Case Study

Our main case study focused on JTOPAS [16], an open source Java library for parsing arbitrary text. We analyzed version 0.4 of the library from the Software-artifact Infrastructure Repository (SIR) [24], which comes with 10 seeded faults. ANGELINA can help diagnose four of these faults. The remaining faults are either not 1-fixable (and there is no obvious way make them so) or they could not be analyzed due to our limited handling of arrays. Table 7 shows a short description of each fault; why it could not be analyzed by the tool; or, if analyzed, what scope and code modifications were applied.

Faults 1, 2 and 6. Table 5 shows the results of applying ANGELINA to JTOPAS faults 1, 2 and 6. The "repair cand." column displays the total number of repair candidates for each fault. The "time" column specifies the running time in seconds; "loaded code" shows the number of classes and methods loaded by JPF; "instructions" is the number of executed bytecode instructions; and "heap" shows the number of new objects created by JPF's virtual machine while executing the system under test. Multiple rows of data for fault 6 indicate that its detection required repeated application of the tool. We show the results for each application separately, starting with the top-level scope.

Faults 1 and 2 were minor, and their details are fully covered by the description in Table 7. Fault 6, however, was intricate, and it involved a large amount of code. We therefore analyzed it in a modular fashion—that is, one method at a time—starting with the top-level test method shown in Fig. 6.

The top-level method tests the JTOPAS Java and Javadoc tokenizers by checking that their outputs satisfy two properties: (1)

fault	repair cand.	time (sec)	loaded code classes	loaded code methods	executed instructions	heap
1	2	2	98	1259	6,063	940
2	2	1	98	1,258	4,052	539
6	13	45	140	2,094	300,219	2,864
	4	29	140	2,093	312,430	4,955
	1	24	139	2,091	257,847	4,778

Table 5: The results of applying ANGELINA to three known faults in JTOPAS.

```
1  public void testJavaTokenizer() throws Throwable {
2    InputStreamTokenizer javaTokenizer =
3      new InputStreamTokenizer(m_reader);
4    InputStreamTokenizer docTokenizer =
5      new InputStreamTokenizer(m_reader);
6    AbstractTokenizer currTokenizer = javaTokenizer;
7    Object openBlock = new Object();
8    Object closeBlock = new Object();
9    Object atSign = new Object();
10
11   int blockBalance = 0;
12   Token token;
13   javaTokenizer.addTokenizer(docTokenizer);
14
15   javaTokenizer.addSpecialSequence("/**", docTokenizer);
16   javaTokenizer.addSpecialSequence("{", openBlock);
17   javaTokenizer.addSpecialSequence("}", closeBlock);
18   docTokenizer.addSpecialSequence("*/", javaTokenizer);
19   docTokenizer.addSpecialSequence("@", atSign);
20   // ...
21   docTokenizer.addKeyword("author");
22   // ...
23
24   while (currTokenizer.hasMoreToken()) {
25     Token token = currTokenizer.nextToken();
26     switch (token.getType()) {
27       case Token.SPECIAL_SEQUENCE:
28         if (token.getCompanion() instanceof AbstractTokenizer) {
29           AbstractTokenizer tokenizer =
30                 (AbstractTokenizer)token.getCompanion();
31           currTokenizer.switchTo(tokenizer);
32           currTokenizer = tokenizer;
33         } else if (token.getCompanion() == openBlock) {
34           blockBalance++;
35         } else if (token.getCompanion() == closeBlock) {
36           blockBalance--;
37         } else if (token.getCompanion() == atSign) {
38           Token token = currTokenizer.nextToken();
39           assert token.getType() == Token.KEYWORD;
40         }
41         break;
42     }
43   }
44   assert blockBalance == 0;
45   // ...
46 }
```

Figure 6: JTOPAS test that exposes Fault 6.

(a) Failing input (b) Passing input

Figure 7: Inputs for the test in Fig. 6.

the '@' token is followed by a keyword inside of a Javadoc comment[1] (line 39), and (2) the number of open braces is the same as the number of closed braces (line 44). The tokenization process has two modes: the Java parsing mode, handled javaTokenizer (line 2), and the Javadoc parsing mode, handled by docTokenizer (line 4). We start in the Java mode (line 6) and then switch (lines 27-32) between the two modes depending on the consumed tokens. The '/**' token triggers the switch from Java to Javadoc (line 15), and '*/' does the opposite (line 18).

When testJavaTokenizer is executed on the Java snippet[2] in Fig. 7(a), it fails with an assertion violation on line 39. In particular, the 'author' token that follows the '@' character is not interpreted as a keyword, against expectation. Executing the test on the snippet in Fig. 7(b) results in normal termination. Given these snippets as inputs, and the while loop in the test method as the scope, ANGELINA produces the repair candidates shown in Table 6. The candidates inhibited by the passing test are shaded in gray.

According to the results in Table 6, the failing test can be rescued in essentially one of three ways: (1) pretend that the first read is not '/**' but some other (non-special) string; (2) pretend that the second token is not the '@' character; and (3) pretend that the third token ('author') has the type KEYWORD. The first fix could be applied on lines 25, 26, 28 and 30 of the first iteration, causing the tokenization process to remain in the Java mode. Since the '@' character has no meaning in the Java mode, it is not associated with the atSign object, and the conditional on line 37 would evaluate to false. This fix, however, is ruled out by the passing test. If the passing test remains in the Java mode, by interpreting the first token as anything other than '/**' in the passing test, the execution will always violate the balanced braces property on line 44.

The second fix—pretending that '@' is something else—can be applied in the second iteration, on lines 25, 26, 28, 30 and 37. It rescues the test case in the same way as the first fix, by ensuring that the conditional on line 37 evaluates to false. Like the first fix, the second fix is essentially equivalent to changing the input, which is not an acceptable solution. The third fix suggests that the program could be rescued by changing the output of nextToken on line 38 to produce a token of type KEYWORD. This is, in fact, the real fix.

Given the buggy call to nextToken as the scope, ANGELINA returns four repair candidates. (For reference, nextToken consists of 80 lines of code.) Three of the candidates simply replace the return value of the method with an angelic token that has the right type, essentially ignoring the computation of the next token. The remaining candidate suggests calling the method test4Normal with a different input token. When the analysis is scoped to test4Normal, the tool pinpoints the seeded faulty expression as the only repair candidate.

5. RELATED WORK

Fault Localization by Altering Program States. Cleve and Zeller [7] propose *cause transitions* as being indicators of potentially faulty

[1]Recall that a Javadoc comment is a block comment that starts with the sequence '/**'.

[2]The test input we retrieved from SIR was too large for our infrastructure to handle, so we manually minimized it to the snippet shown here.

line	itr.	code
25	1	Token token = **currTokenizer.nextToken()**
25	2	Token token = **currTokenizer.nextToken()**
26	1	switch (**token**.getType())
26	2	switch (**token**.getType())
28	1	if (**token**.getCompanion() . . .)
28	2	if (**token**.getCompanion() . . .)
30	1	(AbstractTokenizer) **token**.getCompanion()
37	2	else if (**token**.getCompanion() == atSign)
37	2	else if (token.getCompanion() == **atSign**)
38	2	Token **token** = currTokenizer.nextToken()
39	2	**token**.getType() == Token.KEYWORD
39	2	token.getType() == **Token.KEYWORD**
39	2	**token.getType() == Token.KEYWORD**

Table 6: Repair candidates for testJavaTokenizer. Gray shading highlights the candidates inhibited by the passing input.

statements. A *cause* is defined as the minimal part of the program state, which when changed, can change a passing execution into a failing one. To find the cause, their technique compares program states of a failing execution with those of a neighboring passing execution at comparable stages in the two executions. At each such comparable stage, the technique finds the cause by using ideas from delta minimization algorithm [30]. That is, it finds the smallest state that can be spliced from the state of the failing execution into the state of the passing execution such that the passing execution no longer passes. Causes thus identified can give valuable debugging clues. More importantly, program points at which cause changes from one variable (or a set of variables) to another—cause transition points—are indicative of faulty code.

Jeffrey et al. [12] present a somewhat similar idea, in that they too splice in interesting values into an execution to impact its outcome. However, they splice values from a "value profile" into one statement of a failing execution to see if the failing run can change into a passing one. In contrast to Cleve and Zeller [7], the value profile need not arise from state of any single nearby passing run, rather, it can be based on profiling a mix of program executions. Statements that admit successful value replacements indicate faulty code, but are not necessarily the actual fault. A ranking heuristic is therefore used to order this set of statements on suspiciousness. Statements that are associated with successful value replacements in the greatest number of faulty runs are deemed to be the most suspicious. This work is a generalization of earlier work by Zhang et al. [32], and Wang and Roychoudhury [26], in which only predicate statements were considered for alteration. We extend this line of work by adding the crucial test of flexibility, which determines whether an expression permits multiple value in *passing* tests.

We can think of obtaining values from one run to splice into the other as a heuristic implementation of the angel. By contrast, our implementation of the angel is based on symbolic execution, but the underlying purpose is to find values which lead to successful execution. Our technique does not rely on another execution as a source of values. However, a symbolic implementation of an angel is more expensive computationally. Also, our implementation lets us find repair candidates in heap manipulating programs. Since addresses cannot be spliced across runs, none of the previous techniques has been shown to work with heap bugs. This is a key advantage of our approach, and one that is very pertinent to Java bugs.

Mutation Testing. The goal of mutation testing [8] is to enhance the adequacy of a passing test suite for a program. The basic idea is this: suppose we change ("mutate") some expression in the pro-

Fault number	Description of the fault	ANGELINA's diagnosis	Scope and code modifications
Fault 1	In a constructor of ExtIOException, call super(fmt) is commented out. fmt initializes a format field	ANGELINA offers to match up a read of format field to the expected value in the test harness. This is not the real fix since the program is not 1-fixable	Changed type of format to Object, as ANGELINA cannot handle strings (arrays). Scope was the test harness code
Fault 2	In a constructor of ExtIOException, the boolean field isWrapper is assigned the opposite value	ANGELINA can find the exact expression to be fixed	No modifications. Scope was the class ExtIOException
Fault 3	The virtual method getMessage is commented out, causing the base class's implementation to be used	ANGELINA cannot find the fault. The test harness is heavily dependent on string manipulation	
Fault 4	Off-by-one error in string copy	ANGELINA cannot find the fault as it cannot handle strings (arrays)	
Fault 5	Similar to Fault 4		
Fault 6	Missing setter of type of a Token	As discussed in the text, ANGELINA is able to assist in diagnosing the fault	Reinstated a setter, but changed argument to an incorrect token type. Scope set hierarchically (see text)
Fault 7	Multiple integer constants changed	ANGELINA cannot diagnose the fault	
Fault 8	Similar to Fault 7		
Fault 9	Similar to Fault 3		
Fault 10	Similar to Fault 1		

Table 7: A table of seeded faults in jtopas 0.4. SIR.

gram. These mutations are drawn from a language-specific set of commonly occurring errors. Do we see any failures in the working test suite? If yes, the test suite is powerful enough to kill that "mutant". If not, the programmer tries to create a new test input that can kill that mutant; though, it can sometimes happen that no test input can kill a mutant because that mutant happens to result in a semantically equivalent program. By carrying out mutations at different points in a program, and then trying to kill those mutants, a programmer can increase confidence in the test suite.

Debugging is a closely related problem. In testing, the goal is to find a test input that kills a mutant. In debugging, the goal instead is to find a mutant that passes a new test input that is failing before mutation, and just as importantly, that is *not* killed by existing test cases. In the terminology of this paper, the mutation we seek must be on a flexible expression, because otherwise it would be killed by an existing test case. This is the role of the flexibility check. The more comprehensive a set of passing test cases, the more expressions can be found to be *in*flexible, leaving aside only the expressions at which the program can be fixed.

Note that our current focus in angelic debugging is only on the location of the desired mutant, and not on how to create the mutant in terms of mutation operations. Indeed, not all bugs can be corrected by a standard set of mutations, so we are unable to simply try all mutations of a repair candidate on all passing test cases.

Using Slicing For Debugging. A large body of work exists on debugging based on static and dynamic slicing (e.g. [2, 33, 29]). The basic intuition is that the backward slice with respect to the statement that manifests the bug often contains the cause of the bug. Two ideas on slicing bear close relation to our work.

Dicing is among the first ideas to leverage information from passing tests to help with debugging of failing tests [1]. In fault localization based on dices, one computes portions of slices that appear in failing inputs but not on passing inputs.

Critical slicing [9], which also relates to mutation testing, uses a specific kind of mutation—statement deletion—to determine which statements have a bearing on the final outcome in terms of values of selected output variables. If the value of those output variables does not change even after the statement is deleted, the statement is not critical. For assignment statements, critical slicing checks whether changing $t = e$ to $t = t$ can possibly impact execution of a failing test case. Our technique performs (at higher cost) a more general, existential check: could there exist an e' such that replacing the assignment with $t = e'$ can fix the program.

Fault Localization by Comparing Executions. Several researchers have proposed techniques to use passing runs to localize faults in failing runs. The Tarantula technique [14, 13] compares statement coverage spectrum taken from a number of passing runs, to the coverage obtained from a failing run. This works on the empirical observation that fault locations are correlated with statements that are much more likely to appear in the failing run that in a passing one. In other words, the statement coverage profile of the failing run is an *anomaly* [31]. Anomalies can be detected on other kinds of "signatures" of program executions. For example, Liblit et al. [19] show excellent correlation between faults and anomalous return values of functions, whereas Hangal and Lam [11] detects anomalies on discovered invariants. Reps et al. [23] solves the Y2K problem by comparing executions executed before and after the critical date, both summarized with path profile spectra.

Others have focused on finding "nearby" passing runs to compare to a failing run, with the intention of getting more precise fault localization. For example, Renieris and Reiss [22] select a passing run based on minimizing a nearest-neighbor similarity metric with respect to a failing run; and Artzi et al. [3] use mixed symbolic and concrete execution to generate nearby passing test cases.

Comparing passing and failing executions have also been used in explaining counterexamples in model checking. In works of Ball et al. [4], and Groce and Visser [10], the idea is to compare a single passing trace with a failing trace, and find program points at which the two begin to diverge.

Our work shares with these techniques the intention of getting useful information from passing runs. However, the goal of our work is to *pinpoint* expressions that are good repair candidates. Our technique is geared to answering the question: can this expression be changed to fix the program, given the failing test input as well as the passing ones? In that, the information that we compute is at a much finer granularity. Even if anomaly is perfectly correlated with the real defect, it does not tell us what should be changed in the program. On the other hand, these techniques do have an advantage of being applicable easily to large applications.

Automated Repair. Although the work presented in this paper is about finding repair candidates, and not about finding replacement expressions, there is promising recent work that raises hope that the latter can be automated. We mention four recent lines of research in automated repair. Weimer et al. [28] have shown promising results on automating bug fixing. They use a novel genetic programming algorithm to find suitable replacement expressions to repair bugs. The search for replacement expressions is limited to syntactic constructs found elsewhere in the same program. Program sketching [25] presents another approach to repairing programs in certain domains, when the range of replacement expressions that need to be explored is predetermined, and the computer can simply search for one that passes all test inputs. (Sketching was developed originally for automatically completing partially specified programs, but its applicability to repair is clear.) Malik et al. [20] have presented interesting initial results in repairing heap manipulating programs, for which a representational invariant ("repOK") can be given. The replacement expressions are drawn from a small set of heap manipulating instructions. Finally, Wei at al. [27] automatically generate bug fixes based on deviation from invariants discovered from passing tests; they also use contracts present in Eiffel programs to validate automatically identified program repairs.

6. CONCLUSION

The objective of angelic debugging is to automatically identify expressions in a buggy program that, if changed, can remedy the bug. We observe that an expression is a repair candidate if it can be replaced with a value that fixes a failing test and if, crucially, in each passing test, its value can be changed to another value without breaking the test. We implemented a tool based on this observation on top of the JAVA PATHFINDER model checker. While more experience with angelic debugging is needed, our early experiments suggest that the technique can be useful in speeding up debugging.

7. REFERENCES

[1] H. Agarwal, J. R. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *SRE '95*, 1995.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI '90*, pages 246–256, 1990.

[3] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *ISSTA '10*, pages 49–60, 2010.

[4] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL '03*, pages 97–105, 2003.

[5] R. Bodik, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *POPL '10*, pages 339–352, 2010.

[6] Choco Solver. http://www.emn.fr/z-info/choco-solver, August 2010.

[7] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE '05*, 2005.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[9] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. In *ISSTA '96*, 1996.

[10] A. Groce and W. Visser. What went wrong: explaining counterexamples. In *SPIN'03*, pages 121–136, 2003.

[11] S. Hangal, S. Microsystems, D. C. Shantinagar, and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02*, 2002.

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

[13] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE '05*, pages 273–282, New York, NY, USA, 2005.

[14] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE '02*, pages 467–477, 2002.

[15] Java PathFinder. http://babelfish.arc.nasa.gov/trac/jpf, August 2010.

[16] Java tokenizer and parser tools (JTopas). http://jtopas.sourceforge.net/jtopas/index.html, August 2010.

[17] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS'03*, pages 553–568, 2003.

[18] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[19] B. Liblit, A. Aiken, M. Naik, and A. X. Zheng. Scalable statistical bug isolation. In *PLDI '05*, pages 15–26, 2005.

[20] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid. A case for automated debugging using data structure repair. In *ASE '09*, 2009.

[21] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA '08*, pages 15–26, 2008.

[22] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. *ASE '03*, page 30, 2003.

[23] T. W. Reps, T. Ball, M. Das, and J. R. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *FSE '97*, pages 432–449, 1997.

[24] Software-artifact infrastructure repository (SIR). http://sir.unl.edu/portal/index.html, August 2010.

[25] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS-XII*, pages 404–415, New York, NY, USA, 2006.

[26] T. Wang and A. Roychoudhury. Automated path generation for software fault localization. In *ASE '05*, pages 347–351, 2005.

[27] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *ISSTA '10*, pages 61–72, New York, USA, 2010.

[28] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *CACM*, 53(5), 2010.

[29] M. Weiser. Program slicing. In *ICSE '81*, pages 439–449, 1981.

[30] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE '02*, pages 1–10, 2002.

[31] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[32] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE '06*, pages 272–281, 2006.

[33] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG'05*, pages 33–42, 2005.