SemFix: Program Repair via Semantic Analysis

Hoang D. T. Nguyen Dawei Qi Abhik Roychoudhury
School of Computing, National University of Singapore
{hoangdtn,dawei,abhik}@comp.nus.edu.sg

Satish Chandra
IBM Research
satishchandra@us.ibm.com

***Abstract*—Debugging consumes significant time and effort in any major software development project. Moreover, even after the root cause of a bug is identified, fixing the bug is non-trivial. Given this situation, automated program repair methods are of value. In this paper, we present an automated repair method based on symbolic execution, constraint solving and program synthesis. In our approach, the requirement on the repaired code to pass a given set of tests is formulated as a constraint. Such a constraint is then solved by iterating over a layered space of repair expressions, layered by the complexity of the repair code. We compare our method with recently proposed genetic programming based repair on SIR programs with seeded bugs, as well as fragments of GNU Coreutils with real bugs. On these subjects, our approach reports a higher success-rate than genetic programming based repair, and produces a repair faster.**

I. Introduction

Bug fixing continues to be a mostly manual, time consuming, and therefore expensive activity in software development. Therefore, automated techniques to repair buggy programs can be of tremendous value. In particular, given that compute cycles are cheap and abundant, it makes sense to investigate techniques that help shift the “heavy lifting” of program repair from the human to the computer. While a programmer might not blindly trust a computer-generated fix to her code, her task can become considerably easier: rather than figure out a fix, just verify that an automatically generated fix is correct. Not surprisingly, researchers have recently started looking into automated program repair tools [1]–[3].

We focus on general purpose programs, for which a test suite is available as a way to tell whether the program is working correctly (i.e. it passes all the tests) or not (i.e. there exists a failing test), but otherwise no formal specification of correct behavior is available; this is generally the case in practice (by contrast, kernels that manipulate data structures often do have specifications, and automatic repair on data structure programs have been well studied, for example see [4], [5]). A successful repair would be a modification of the program such that it passes all the tests in the test suite.

One of the most successful techniques in recent work that works on general programs is based on syntactic search. The premise behind this technique is that, once we know where the defective expression is in the program, a correct expression may be present syntactically at another place in the program, so it is a matter of searching over a space of replacements from among existing expressions.[1] The technique uses genetic programming technique for searching over this space, and has been shown to work for large programs [6]. The limitation of this technique is that the correct expression should be present in the program; the technique cannot “synthesize” an appropriate expression from variables and constants.

An obvious response to the limitation would be a search over a space of syntactic expressions, without consideration of whether those expressions appear elsewhere in the program. Such an approach would be more in the flavor of *sketching* [7], [8]. However, unless the space of repair expression is fixed upfront (possibly as a set of templates), such a technique will not work. Furthermore, as our experiments show, enumerating over the set of possible repair templates is inefficient.

In this paper, we explore a constraint based semantic approach towards program repair. The repair constraints are generated by our desire to have the repaired program pass the given test cases. Thus, given a program location to be fixed, we derive constraints on the expression to appear in the program location, in order to have the changed program pass all the given tests. The repair constraints are generated via (controlled) symbolic execution and the expression to be repaired is obtained via program synthesis. We report that, for certain kinds of program and bugs, the semantics-based approach can not only have a higher-success rate than a syntactic search-based approach, but also be able to produce a repair faster. At the same time, we do believe that symbolic execution imposes certain scalability limitations on the size of programs we can handle.

Our approach is a combination of three existing techniques.

- *Fault isolation*, i.e. where to fix the problem. The technique uses the ranking produced by a statistical fault isolation [9] tool (it shares this step with the search-based techniques.) Our approach examines *one buggy statement at a time* from a ranked suspicion report of statements.
- *Statement-level specification inference*. We automatically discover the correct specification of the buggy statement. We use an idea similar to the one used in angelic debugging [10] in converting an expression to a non-deterministic expression. This step allows us to create, for each input to the buggy statement, the output that would have resulted in the test passing.
- *Program synthesis*. The third idea is to use component-based synthesis idea [11] to synthesize an expression that conforms to the specification discovered before.

The inter-play of the second and third steps is the primary novelty of our repair tool. The statement-level specification narrows the search space significantly, and sets up the problem

[1] This is an oversimplification, but broadly speaking this is the idea.

for component-based synthesis that is based on constraint solving. We also apply two other important performance optimizations. First, instead of creating a constraint over the entire test suite, we create it using a subset of tests, adding tests incrementally to the mix. Second, among the set of possible expressions that can be synthesized, we explore in order of increasing complexity, so the tool can find simple fixes quicker.

We have experimentally evaluated our method as well as genetic programming on SIR subjects with seeded bugs, as well as fragments of GNU Coreutils with real bugs. The use of state-of-the-art SMT solvers and program synthesis engines allow our repair timings to be less than that of genetic programming based repair, on these subjects. For the programs with seeded bugs, our repair method can repair three times as many buggy versions as compared to genetic programming. For the Coreutils programs with real bugs, our repair method took an average of 3.8 minutes while genetic programming took an average of 6 minutes.

II. Overview

Given a buggy program P and a test suite T containing at least one failing test case, our program repair technique works as follows. First, we employ statistical fault localization to generate a list L of program statements ranked by their suspiciousness of being the bug. Our core program repair method then scans through the statement list from the most suspicious one to the least suspicious one until a successful repair is generated. For each scanned statement s, our core program repair method tries to repair the program by altering statement s. Assuming that statement s is the root-cause of the failure, our core repair method consists of two major steps: i) we first generate the repair constraint that has to be satisfied by a successful repair on s and ii) we try to solve the repair constraint using program synthesis.

```
 1 int is_upward_preferred(int inhibit, int up_sep,
       int down_sep) {
 2   int bias;
 3   if(inhibit)
 4     bias = down_sep; //fix: bias=up_sep+100
 5   else
 6     bias = up_sep;
 7   if (bias > down_sep)
 8     return 1;
 9   else
10     return 0;
11 }
```

Fig. 1. Code excerpt from Tcas

We illustrate our technique using the sample program in Fig. 1. It is a code excerpt taken from Tcas, traffic collision avoidance system [12]. Suppose inhibit has only two allowable values ({0,1}). The intended behavior of this program is described as follows: `is_upward_preferred = (inhibit*100 + up_sep > down_sep)`. A test suite for checking the correctness of the program is presented in Table I. The testing result shows that the implementation is buggy as two tests are failed. To repair this program, we first employ

TABLE I
A test suite for the program in Fig. 1

Test	Inputs			Expected output	Observed output	Status
	inhibit	up_sep	down_sep			
1	1	0	100	0	0	pass
2	1	11	110	1	0	fail
3	0	100	50	1	1	pass
4	1	-20	60	1	0	fail
5	0	0	10	0	0	pass

TABLE II
Tarantula fault localization result on the program in Fig. 1

Line	Score	Rank
4	0.75	1
10	0.6	2
3	0.5	3
7	0.5	3
6	0	5
8	0	5

Tarantula fault localization [13] for the buggy program using the given test suite. Tarantula is a statistical fault localization technique that is explained in details in Section III-A. Basically, it ranks program statements in a descending order of their suspiciousness. A statement exercised by more failing tests and fewer passing tests will have a higher suspiciousness score. The result of Tarantula shown in Table II is a list of statements ranked by their suspiciousness score. Since line 4 is ranked at the top, we start investigating whether the program can be repaired by changing line 4.

Suppose we want to replace line 4 with $bias = f(\ldots)$, where $f(\ldots)$ is to be figured out by our method. There are four accessible variables at line 4: inhibit, up_sep, down_sep and bias. Since variable bias is not initialized, we assume that it cannot be used by $f(\ldots)$. Let us now assume the signature of function f to be:

```
int f(int inhibit, int up_sep, int down_sep)
```

We then find the constraint that has to be satisfied by $f(\ldots)$ to pass all the test cases in the test suite. This is achieved through symbolic execution.

For each test exercising line 4, we generate one constraint on $f(\ldots)$ whose satisfaction guarantees that the fixed program produces the expected output. We use the second test in Table I to explain how such a constraint is generated. Fig. 2 presents the symbolic execution tree for test 2, with input vector $\langle 1, 11, 110\rangle$. Note that our symbolic execution does not start with program input. Instead, the program is executed concretely with input $\langle 1, 11, 110\rangle$ until it reaches line 4. Then the value of variable bias is replaced with a symbolic value X and the execution continues symbolically. On executing the branch at line 7, the execution is faced with two choices and both paths are executed as shown in the symbolic execution tree. As we know the expected return value for input $\langle 1, 11, 110\rangle$ should be 1, only the path through line 8 should be followed to make the program pass. To follow this path, the path condition, $X > 110$, should be satisfied. Given that the program state at line 4 is {`inhibit == 1, up_sep == 11, down_sep == 110`}, we know that f has to satisfy $f(1, 11, 110) > 110$. Similarly, we get

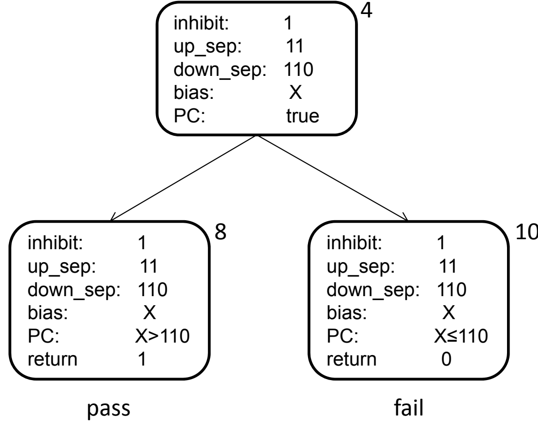

Fig. 2. Symbolic execution tree for test case 2 in Table I when trying to fix line 4 of the program in Fig. 1. Each box denotes a program state at the line annotated in the superscript. The program state includes all values of program variables as well as the path condition PC.

$f(1,0,100) \leq 100$ from test 1 and $f(1,-20,60) > 60$ from test 4. Therefore, the constraint that f needs to satisfy is $f(1,11,110) > 110 \wedge f(1,0,100) \leq 100 \wedge f(1,-20,60) > 60$. We employ program synthesis to solve the constraint for f in order to get a concrete function. Program synthesis requires basic components (e.g. constants, "+", "−") as ingredients to construct the function f. In our technique, these components are incrementally provided to program synthesis. In the first trial, only a constant is allowed. However, no constant function can satisfy the above constraint. We then allow function f to use one "+", i.e. f can take either the form of $var_1 + c$ or $var_1 + var_2$, where var_1 and var_2 are in {`inhibit`, `up_sep`, `down_sep`} and c is an integer constant. The synthesis procedure can find a solution $f(inhibit, up_sep, down_sep) = up_sep + 100$ which is a successful repair to the program in Fig. 1. Note that if "−" is used instead of "+", we will get $f(inhibit, up_sep, down_sep) = up_sep - (-100)$ as repair.

III. Background

A. Statistical Fault Localization

Statistical fault localization [9], [13] aims to localize the root-cause of a program failure by exploiting the correlation between execution of the faulty statements and program failure. A suspiciousness score is computed for each program statement based on its frequency of occurrence in passing and failing executions. Based on the suspiciousness score, a ranked list of statements is given to users. Users can then examine the ranked list from the most suspicious statement to the least suspicious statement until the failure root-cause is found.

In this paper, we adopt the suspiciousness score from Tarantula technique [13]. For a statement s, its suspiciousness score $susp(s)$ is computed as

$$susp(s) = \frac{failed(s)/totalfailed}{passed(s)/totalpassed + failed(s)/totalfailed}$$

where $failed(s)$ denotes the number of failing executions in which s occurs and $passed(s)$ denotes the number of passing executions in which s occurs. The variable $totalfailed$ denotes the total number of failing executions and $totalpassed$ denotes the total number of passing executions.

B. Component-based Program Synthesis

We briefly introduce the recent advance in component-based program synthesis [11]. Given a set of input-output pairs, component-based program synthesis generates a program that satisfies all the given input-output pairs. More specifically, if $\langle \alpha, \beta \rangle$ is one of the input-output pairs, then the synthesized program must produce output β when its input is α. In component based program synthesis, we provide a set of basic components that the to-be-synthesized function f is allowed to use. For example, to synthesize a program with linear expressions, {constant, minus, plus} are given as the basic components. A set of location variables are defined for each component and the synthesis process is reduced to finding values for these location variables. The constraint over the location variables is in first-order logic and solved by an SMT solver. If it has a solution, a unique program can be constructed based on the values of location variables. We now explain the encoding method.

Suppose we provide N components $\{f_1, \ldots, f_N\}$ to synthesize function f. Without losing generality, we assume each component only has one output. For the ith component, we denote its input as $\overrightarrow{\chi}_i$ and its output as r_i. We use Q to denote the set of all input variables from all components and R to denote the set of output variables from all components.

$$Q := \cup_{i=1}^{N} \overrightarrow{\chi}_i \qquad R := \cup_{i=1}^{N} \{r_i\}$$

We use $\overrightarrow{\chi}$ to denote the input variables for function f and use r to denote the output variable of f. The set of location variables is defined as

$$L := \{l_x | x \in Q \cup R \cup \overrightarrow{\chi} \cup \{r\}\}$$

A location variable l_x denotes where variable x is defined. Given a valuation of L, a program can be constructed using the following procedure $Lval2Prog(L)$. Here the ith line of the constructed program is $r_j = f_j(r_{\sigma(1)}, \ldots, r_{\sigma(\eta)})$ when $l_{r_j} == i$ and $\wedge_{k=1}^{\eta}(l_{\chi_j^k} == l_{r_{\sigma(k)}})$, where η is the number of inputs for component f_j and χ_j^k denotes the kth input parameter of component f_j. The program output is produced in line l_r.

We use one example below to explain the meaning of location variable. Suppose we only provide one component +, whose inputs are χ_+^1, χ_+^2. Since there is only one component +, we use + instead of its component number as the subscript for clarity. The output variable for + is r_+. Suppose there is only one input for the synthesized program. Let the value of location variables be $\{l_{r_+} == 1, l_{\chi_+^1} == 0, l_{\chi_+^2} == 0, l_r == 1, l_{\chi^1} == 0\}$. Given that $l_{r_+} == 1$, r_+ is defined in line 1 and thus component + is placed in line 1. If the location variables $l_{\chi_+^1} == l_{\chi_+^2} == 0$, then both χ_+^1 and χ_+^2 are the same as the variable defined in line 0, which means that they are the same as the output of line 0. Since l_r equals 1, the value defined at line 1, $\mathtt{r}_+$, is the output of the program. From the valuation of the location variables, we can construct the following program

```
0 r₀ = input⁰;
1 r₊ = r₀ + r₀;
2 return r₊;
```

where $\texttt{input}^0$ denotes the first input parameter of the synthesized program.

The location variables have to satisfy certain constraints so that the corresponding program can pass on all given input-output pairs. We first give the well-formedness constraint ψ_{wfp}. Let $M = |\overrightarrow{\chi}| + N$, where N is the number of components provided to the program synthesis procedure.

$$\psi_{wfp}(L,Q,R) \stackrel{\text{def}}{=} \bigwedge_{x\in Q}(0 \le l_x < M) \wedge \bigwedge_{x\in R}(|\overrightarrow{\chi}| \le l_x < M) \wedge \psi_{cons}(L,R) \wedge \psi_{acyc}(L,Q,R)$$

$$\psi_{cons}(L,R) \stackrel{\text{def}}{=} \bigwedge_{x,y\in R, x\not\equiv y}(l_x \ne l_y)$$

$$\psi_{acyc}(L,Q,R) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{N} \bigwedge_{x\in\overrightarrow{\chi_i}, y\equiv r_i} l_x < l_y$$

The constraint ψ_{cons} dictates that there is only one component in each line and ψ_{acyc} encodes that inputs of each component are defined before they are used.

The constraint ψ_{wfp} only guarantees that L corresponds to a well-formed program f. We use the following constraint to guarantee that the solution f satisfies all the given input-output pairs.

$$\phi_{func}(L,\alpha,\beta) \stackrel{\text{def}}{=} \psi_{conn}(L,\overrightarrow{\chi},r,Q,R) \wedge \phi_{lib}(Q,R) \wedge (\alpha = \overrightarrow{\chi}) \wedge (\beta = r)$$

$$\phi_{lib}(Q,R) \stackrel{\text{def}}{=} (\bigwedge_{i=1}^{N} \phi_i(\overrightarrow{\chi}_i, r_i))$$

$$\psi_{conn}(L,\overrightarrow{\chi},r,Q,R) \stackrel{\text{def}}{=} \bigwedge_{x,y\in Q\cup R\cup\overrightarrow{\chi}\cup\{r\}} (l_x = l_y \Rightarrow x = y)$$

The semantics of each basic component is encoded into ϕ_{lib} with ϕ_i representing the specification of component f_i and the relation between location variables and program variables is encoded in ψ_{conn}. Collectively, $\phi_{func}(L,\alpha_i,\beta_i)$ represents that when executing the synthesized function f with input α_i, the output should be β_i.

$$\theta \stackrel{\text{def}}{=} (\bigwedge_{i=1}^{n} \phi_{func}(L,\alpha_i,\beta_i)) \wedge \psi_{wfp}(L,Q,R)$$

Finally, given n input-output pairs $\{\langle\alpha_i,\beta_i\rangle | 1 \le i \le n\}$, the constraint θ represents that the synthesized function f should satisfy all input-output pairs and the function has to be well-formed. Given a solution L_0 to the constraint θ, we can construct a program that satisfies all the I/O pairs $\langle\alpha_i,\beta_i\rangle$ using the aforementioned procedure $Lval2Prog$.

IV. Detailed Methodology

We only generate repairs by altering one statement. The generated fix is always with respect to a given test suite. If all the tests in the given test suite pass after the repair is applied, we consider the repair to be successful. We now elaborate on the different steps of our method.

A. Generating Repair Constraint

In this paper, we focus on repairs that change the right hand side of assignments or branch predicates. We do not generate any repair that require changing the left hand side of an assignment statement. Albeit simple, we will show how common programming bugs can be fixed by altering only branch predicates or the right hand side of assignments.

The repair generated by our method has one of the following two forms:

- x = $f_{buggy}(\ldots) \rightarrow$ x = $f(\ldots)$
- if($f_{buggy}(\ldots)$) $\rightarrow$ if($f(\ldots)$)

In either case, we generate an expression $f(\ldots)$ that is used to replace either the right hand side of an assignment or a branch predicate. We require the expression $f(\ldots)$ to be side-effect free. That is, no program variable is modified when $f(\ldots)$ is evaluated. In the remainder of this paper, we do not distinguish between these two cases unless necessary. An expression is in essence a function. For example, the expression $x + y$ can be treated as the function $f(x,y) = x + y$. For ease of presentation, we use function instead of expression in the rest of this paper.

Definition 1 (Repair Constraint): Given a program P, a test suite T, a repair constraint C of a function f_{buggy} in program P is a constraint over function f such that if $f \models C$, $P[f/f_{buggy}]$ passes all tests in T.

Given a suspicious statement s that contains a buggy function f_{buggy}, we now explain how to generate the repair constraint C such that if a function f satisfies C, the program can be repaired by replacing f_{buggy} with f. As mentioned, the concept of repair is with respect to the given test suite T. The repair constraint C is a conjunction of constraints that are derived from T. Suppose the test suite has n tests, $T = \{t_i | 1 \le i \le n\}$, which essentially are different input vectors for running P. The repair constraint is $C = \bigwedge_{i=1}^{n} C_i$. For each test t_i, we show in the rest of this section how to generate a constraint C_i such that if the function f satisfies C_i, the program generated by replacing f_{buggy} with f passes for test t_i.

Each C_i is a second order predicate over the function f. To generate C_i, we use symbolic execution in a novel fashion. Before we elaborate our special symbolic execution, let us briefly recap normal symbolic execution [14], [15]. Traditionally, symbolic execution executes a program by considering all input variables as symbolic. The state of each variable during symbolic execution is represented as an expression of the symbolic inputs. When a branch is executed, if both directions of the branch are feasible, both paths are executed by the symbolic execution. For each executed path, a path condition is collected. The path condition of a path π is a predicate on the program inputs, and any input satisfying the path condition of path π follows path π. The output of the symbolic execution is a set of feasible paths and the corresponding path conditions

of these paths. For each path, we can also get the symbolic representation of the program output in terms of the program input variables.

Different from traditional symbolic execution, our symbolic execution starts with a concrete input t_i. Suppose the program statement we try to fix is s. Let us assume that s is executed at most once during one execution for the moment. We execute the program *concretely* with input t_i to statement s (without executing statement s). We denote the program state before executing statement s as ξ_i. Then we set the result of function $f(\ldots)$ as symbolic and continue *symbolic execution* from statement s. We use τ_i to denote the symbolic value assigned to the result of function $f(\ldots)$. Suppose the symbolic execution explores m paths. For each explored path π_j, $1 \leq j \leq m$, we denote the associated path condition as pc_j and the symbolic expression of the output as O_j. We use $O(t_i)$ to denote the expected output of program P with input t_i. The constraint C_i is

$$C_i := (\bigvee_{j=1}^{m} (pc_j \land O_j == O(t_i))) \land (f(\xi_i) == \tau_i)$$

The first part of C_i, $\bigvee_{j=1}^{m}(pc_j \land O_j == O(t_i))$, dictates that there is at least one feasible path (as guaranteed by the satisfiability of path condition) along which the output of program P is the same as the expected output $O(t_i)$. The second part of C_i, $(f(\xi_i) == \tau_i)$, builds up the input-output relationship of function f. With program state ξ_i as input, the output of function f, τ_i, has to satisfy $\bigvee_{j=1}^{m}(pc_j \land O_j == O(t_i))$. When statement s is not exercised during executing program P with input t_i, C_i is *true* if the execution passes and *false* if the execution fails.

We use the same example in Section II to explain the generation of C_i. Take the second test from Table I for the program in Fig. 1. The symbolic execution tree for this test case has been given in Fig. 2. We use X to denote the symbolic variable to be consistent with Section II. The symbolic execution explores two paths: i) path condition is $X > 110$ and output is 1, and ii) path condition is $X \leq 110$ and output is 0. Given the output of expected output of the test is 1, the first part of C_i is then $(X > 110 \land 1 = 1) \lor (X \leq 110 \land 0 = 1)$, which can be simplified to $X > 110$. Given the program state at the fix location is {`inhibit == 1, up_sep == 11, down_sep == 110`}, the second part of the C_i is $f(1, 11, 110) = X$. Therefore, the constraint C_i is $f(1, 11, 110) > 110$.

After generating the constraint C_i from each test t_i, the full constraint to be solved is

$$C := \bigwedge_{i=1}^{n}((\bigvee_{j=1}^{m} (pc_j \land O_j == O(t_i))) \land (f(\xi_i) == \tau_i))$$

The solution of C is a function f which can be used to repair the program so that all the test cases in the test suite T pass.

Note that C is a constraint over function f. Current off-the-shelf SMT solvers cannot directly solve such a formula. We will explain in Section IV-B how we solve C indirectly through program synthesis. One special case is when function f is a constant function. In this case, we can replace $f(\ldots)$ with a free variable and constraint C becomes a first-order constraint that can be solved by current SMT solvers.

Repair a statement that is executed multiple times: If the repaired statement is executed more than once (either inside a loop or inside a function that is called multiple times) during the execution of test t_i, we use τ_i^k to represent the value produced by $f(\ldots)$ in the kth time it is executed. We use ξ_i^k to represent the program state before the kth time f is executed when executing program P with input t_i. Note that the variable values in ξ_i^k could be symbolic in terms of $\{\tau_i^1, \ldots, \tau_i^{k-1}\}$. Suppose statement s is executed w times during symbolic execution. We have

$$C_i := (\bigvee_{j=1}^{m} (pc_j \land O_j == O(t_i))) \land (\bigwedge_{k=1}^{w} f(\xi_i^k) == \tau_i^k)$$

$$C := \bigwedge_{i=1}^{n} C_i$$

Now each symbolic output O_j is an expression in terms of $\tau_i^1, \ldots, \tau_i^w$. The second part of C_i becomes $(\bigwedge_{k=1}^{w} f(\xi_i^k) == \tau_i^k)$, which means that the input-output relationship of function f has to be satisfied each time f is executed. During the kth time f is executed, with program state ξ_i^k, the output of the function f is τ_i^k.

Infinite loop in symbolic execution: If the termination condition of a loop is an expression over our introduced symbolic variables, the symbolic execution may never terminate exploring infinite number of loop iterations. Suppose we have `while(i<x){x=buggy-expression;}`. Each time the buggy statement is executed, we will assign a new symbolic value to variable x. On checking the loop condition $i < x$ for the next iteration, both directions are feasible since the new symbolic value of x is unbounded. Thus, symbolic execution continues to explore more and more loop iterations and never terminates. To avoid such infinite loop exploration, we set a loop bound B for all loops in symbolic execution. After a loop is iterated B times, the symbolic execution stops exploring the path that leads to the next iteration. Note that this does not break the validity of repair constraint. If the repair constraint C has a solution f, for the sub-constraint C_i, one of the constraint $pc_j \land O_j == O(t_i)$ is satisfied in C_i. It is guaranteed that when applying the repair f, the repaired program with input t_i will follow the path π_j whose path condition is pc_j and generate the correct output. On the other hand, if the loop bound B is too small, we may miss repairs that drive the program to follow paths containing larger number of loop iterations.

B. Generating a Fix

After generating the repair constraint using symbolic execution, we elaborate how to solve the repair constraint in this section. Recall that the repair constraint is

$$C := \bigwedge_{i=1}^{n}((\bigvee_{j=1}^{m} (pc_j \land O_j == O(t_i))) \land (\bigwedge_{k=1}^{w} f(\xi_i^k) == \tau_i^k))$$

To solve the repair constraint C, we leverage the recent advance in component based program synthesis [11].

We have reviewed the core method of component based program synthesis in Section III-B. The input-output pairs of the to-be-synthesized program are encoded into constraints on a set of location variables L, a valuation of which leads to a program that satisfies the given input-output pairs. More specifically, the constraint $\psi_{func}(L, \alpha, \beta)$ dictates that the synthesized program must produce output β when given input α. In our repair constraint, we also have input-output pair $\langle \xi_i^k, \tau_i^k \rangle$ that is generated when f is hit at the kth time in the execution of program P with input t_i. However, $\langle \xi_i^k, \tau_i^k \rangle$ is symbolic in terms of $\{\tau_i^k | 1 \leq k \leq w\}$, where w is the number of times f is executed with input t_i. Moreover, the variables $\{\tau_i^k | 1 \leq k \leq w\}$ have to satisfy $(\bigvee_{j=1}^{m}(pc_j \wedge O_j == O(t_i)))$ to make the execution pass, i.e., program output is as expected. Therefore, the constraint to be satisfied by f during the execution of input t_i is

$$\theta_i \overset{\text{def}}{=\!=} \exists \overrightarrow{\tau_i}, \bigwedge_{k=1}^{w} \phi_{func}(L, \xi_i^k, \tau_i^k) \wedge (\bigvee_{j=1}^{m}(pc_j \wedge O_j == O(t_i)))$$

where $\tau_i := \{\tau_i^k | 1 \leq k \leq w\}$.

Conjoining the constraints from all tests together with the well-formedness constraint ψ_{wfp}, we get the following constraint θ, a solution to which corresponds to a function f that is a valid repair.

$$\theta \overset{\text{def}}{=\!=} (\bigwedge_{i=1}^{n} \theta_i) \wedge \psi_{wfp}(L)$$

It is worth noting that program synthesis is not the only way to solve the repair constraint. Given a set of components, an enumeration on all possible compositions of components could also be used. For example, given {+, -} as the components, we could enumerate all possible expressions that can be constructed using {+, -} and accessible program variables at the repair location. Through enumeration, we try to find a function that satisfies the repair constraint. As shown by Gulwani, et. al. [16], program synthesis is more efficient than enumeration on all possible compositions of the components.

C. Putting it All Together

Our repair algorithm is presented in Algorithm 1. The algorithm takes as inputs a buggy program P, a test suite T and a ranked list of suspicious program statements RC. When successful, our algorithm produces a repair, applying which on P makes P pass all tests in the test suite T.

Our algorithm scans though RC iteratively until a successful repair can be generated. In each iteration, it first takes the most suspicious statement that has not been tried out using Shift(RC). Shift(RC) returns the first element from list RC and removes that element from the list. Ideally, given a repair candidate and the test suite T, we can directly apply the technique presented in previous sections to generate a repair. However, we notice that the test suite could be large and thus affect the scalability of our technique. We present in Algorithm 1 a technique to select a subset of the entire test suite for repair generation. The selection of tests is driven by testing. We use the set S to maintain the tests used in generating the repair constraint. Initially, set S only contains the failing tests in T. After a repair is generated, we test whether any test that is not in S fails in the repaired program. If a test t' fails in the repaired program P', we add t' into set S and re-generate the repair with the augmented set S. Note that this augmentation process is guaranteed to terminate as the maximum size of S is limited by the test suite T. In the worst case, we may need to add all tests in T into S. Nevertheless, as evidenced by our experiments, usually a small number of test cases is sufficient compared to the test suite size.

Algorithm 1 Repair algorithm

```
1:  Input:
2:  P : The buggy program
3:  T : A test suite
4:  RC : A ranked list of potential bug root-cause
5:  Output:
6:  r: A repair for P
7:
8:  while RC is not EMPTY and not TIMEOUT  do
9:      rc = Shift(RC) // A repair candidate
10:     S = ∅         // A test suite for repair generation
11:     T_f = ExtractFailedTests(T, P);
12:     while T_f ≠ ∅  do
13:         S = S ∪ T_f
14:         new_repair = Repair(P, S, rc)
15:         if new_repair == null then
16:             break
17:         end if
18:         P′ = ApplyRepair(P, new_repair)
19:         T_f = ExtractFailedTests(T, P′);
20:     end while
21:     if new_repair not null then
22:         return new_repair
23:     end if
24: end while
25:
26: function Repair(P, S, rc)
27:     C = GenerateRepairConstraint(P, S, rc);
28:     level = 1  // The complexity of a repair
29:     new_repair = Synthesize(C, level);
30:     while new_repair == null and level ≤ MAX_LEVEL do
31:         level = level + 1
32:         new_repair = Synthesize(C, level);
33:     end while
34:     return new_repair
35: end function
```

The *Repair* function in Algorithm 1 tries to generate a repair with a program P, a set of tests S and a repair candidate rc. It first uses symbolic execution to generate the repair constraint using the method presented in Section IV-A. It then employs program synthesis to solve the repair constraint. During program synthesis, we need to provide the set of components that can be used by the repair. In order to reduce the complexity of the generated repair as well as increase the scalability of program synthesis, we categorize common components according to their complexity level and feed them incrementally to the program synthesis procedure. The categorization of basic components is shown in Table III. The synthesis procedure starts with level 1, which means

TABLE III
THE CATEGORIZATION OF BASIC COMPONENTS

Level	Conditional Statement	Assign Statement
1	Constants	Constants
2	Comparison ($>$, $\geq$, $=$, $\neq$)	Arithmetic ($+$, $-$)
3	Logic ($\wedge$, $\vee$)	Comparison, Ite
4	Arithmetic ($+$, $-$)	Logic
5	Ite, Array Access	Array Access
6	Arithmetic ($*$)	Arithmetic ($*$)

that only constants are allowed in the constructed repair. If it fails, more components are provided gradually based on their complexity level. When at level $level$, all the components whose level is less than or equal to $level$ are provided to program synthesis. This process continues until a repair is generated or the MAX_LEVEL is reached.

V. IMPLEMENTATION

We introduce SEMFIX (Semantic-based Program Fixing) tool as an implementation of our technique. The architecture of our tool is presented in Fig. 3. SEMFIX receives a buggy program and a test suite as inputs. As the result, it generates a repair for the buggy program. SEMFIX is the synergy of fault localization, symbolic execution and program synthesis. We adapted the Tarantula technique [13] as mentioned in Section III-A to provide a ranked list of statements according to their suspiciousness score. KLEE [14] is employed for generating repair constraints. KLEE is a static symbolic execution engine which is mainly used for generating high coverage test suites and finding bugs. By default, it uses depth-first-search to explore all program behaviors w.r.t. predefined symbolic variables. In our program repair context, the variable that is directly affected by a potential defect is treated as a symbolic variable. To avoid changing and recompiling program's source code whenever a new potential defect is selected, symbolic variables are instrumented at runtime. At the program location where a new symbolic variable is introduced, the values of all accessible variables are also gathered. To generate the repair constraint, for each explored path, we collect the path condition as well as the symbolic output. The program synthesis module is implemented in Perl, and Z3 SMT solver [17] is used to solve a repair constraint. After a fix is constructed, it is simplified and transformed to make it more readable. For instance, $c == 97$ is transformed into $c ==$ 'a' (c is in `char` type), and $a + a$ is transformed into $2 * a$.

We use the following optimization to avoid program synthesis whenever we can. Suppose we try to generate a repair by modifying statement s. Recall that our repair constraint C is a conjunction of each basic constraint C_i, where

$$C_i := (\bigvee_{j=1}^{m} (pc_j \wedge O_j == O(t_i))) \wedge (\bigwedge_{k=1}^{w} f(\xi_i^k) == \tau_i^k)$$

Let C'_i be $(\bigvee_{j=1}^{m} (pc_j \wedge O_j == O(t_i)))$, which is a first order constraint over the symbolic variables $\{\tau_i^k | 1 \leq k \leq w\}$, where w is the number of symbolic variables. We check whether each C'_i is satisfiable using Z3. If any C'_i is not satisfiable, clearly C_i is unsatisfiable and hence the repair constraint C is unsatisfiable. In that case, we quickly conclude that the statement is not fixable and avoid program synthesis.

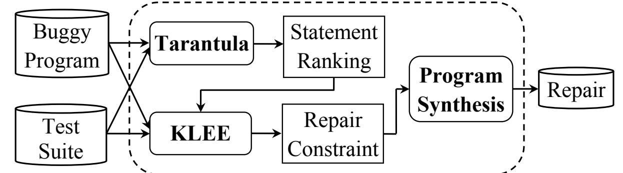

Fig. 3. Architecture of SEMFIX tool.

TABLE IV
SUBJECT PROGRAMS FROM SIR REPOSITORY.

Subject Prog.	Size (LOC)	#Versions	Description
Tcas	135	41	air-traffic control program
Schedule	304	9	process scheduler
Schedule2	262	9	process scheduler
Replace	518	29	text processor
Grep	9366	2	text search engine
Total		**90**	

VI. EMPIRICAL EVALUATION

To assess the efficiency and effectiveness of our technique, we employed SEMFIX tool to repair seeded defects as well as real defects in an open source software. We also compared our method with genetic programming based repair techniques. All the experiments were conducted on a Core 2 Quad 2.83GHz CPU, 3GB memory computer with Ubuntu 10.04 OS.

A. Experience with SIR Programs

Subject programs used in this experiment (see Table IV) are from Software-artifact Infrastructure Repository (SIR) [12]. They represent different kind of applications ranging from a air-traffic control system (Tcas), scheduling utilities (Schedule, Schedule2) to strings and files manipulation programs (Replace, Grep). Each subject program comes with a test suite and has multiple buggy versions as shown in "#Versions" column. Each buggy version has one or many seeded defects that represent common programming errors. Program versions that pass all tests are excluded as there is no witness of failure. Each SIR program has thousands of tests. Normally, a program of similar size to SIR programs does not have this enormous number of tests. Hence, a given test suite is minimized to a set of 50 tests that achieves a maximum (line) coverage.

For SEMFIX, the maximum level (MAX_LEVEL) is set to 3. This level is chosen to keep search space within a reasonable size. Higher levels are only allowed when buggy locations are provided by users. In trying to fix each buggy program, we set the time bound for SEMFIX to 4 minutes. All subject programs but Tcas use only local variables to construct a repair. Global variables are used to construct a repair in Tcas because they appear in most of Tcas computations. The result of SEMFIX on SIR programs are presented in Table V. The time taken for SEMFIX is shown in Fig. 4. The numbers under GP (GenProg) are explained later in Section VI-C.

We have experimented with different test suite sizes. The success rate decreases with more number of tests. Intuitively, it is more difficult to generate a repair to pass more tests. It also

TABLE V
COMPARING THE SUCCESS RATE BETWEEN SEMFIX (SF) AND GENPROG (GP). X IN [X] ON THE TOP OF EACH COLUMN DENOTES THE NUMBER OF TESTS.

Program	[10] SF/GP	[20] SF/GP	[30] SF/GP	[40] SF/GP	[50] SF/GP
Tcas	38 / 24	38 / 19	35 / 16	34 / 12	34 / 11
Schedule	5 / 1	3 / 1	4 / 1	4 / 0	4 / 0
Schedule2	4 / 4	3 / 2	4 / 2	3 / 3	2 / 1
Replace	7 / 6	7 / 5	8 / 5	7 / 6	6 / 4
Grep	2 / 0	1 / 0	1 / 0	2 / 0	2 / 0
Total	56 / 35	52 / 27	52 / 24	50 / 21	48 / 16

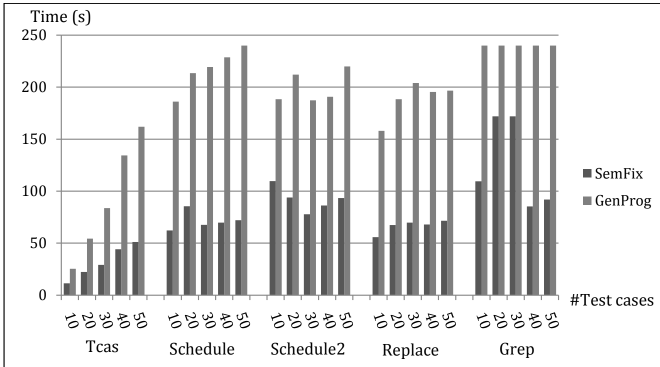

Fig. 4. Comparing the running time between SEMFIX and GenProg.

shows that the repairs generated with small number of tests may not be valid for some other tests that are not in the test suite. We note that this weakness is inherent in any test suite based program repair, since no formal program specification is given and repairs can only be generated with respect to limited number of given tests. Out of the 90 buggy programs, with a test suite size of 50 — SEMFIX repaired 48 buggy programs while genetic programming repaired only 16. The average time required by SEMFIX for each repair is less than 100 seconds.

Repairs generated by SEMFIX*:* SEMFIX was able to correct various types of bugs. Table VI shows the number of successful cases for each type of bug. The number of total bugs of a particular type is specified in the second row of Table VI. *Const* refers to bugs that use the wrong constant. *Arith* refers to using wrong arithmetic expression, and *comp* refers to using the wrong comparison operators. *Logic* refers to using wrong logic operators. Programs with *code missing* bugs lack some statements, and programs with *redundant code* contain some statements that are not supposed to exist.

Repairs that SEMFIX *could not generate:* We have manually looked into the cases where SEMFIX failed to repair the buggy programs. Out of the 90 buggy program versions, SEMFIX failed in 42 of them. The reasons include i) the repairs require a larger loop bound than the bound used in our experiment (8 versions) ii) the repairs require precise modeling of array, which is not fully supported by our current implementation (6 versions) iii) the repairs require symbolic execution with floating-point variables, which is not supported by KLEE (5 versions) iv) the repairs (15 versions) require more than one line fix, 11 of them are from code-missing

TABLE VI
SEMFIX (SF) VS. GENPROG (GP) IN REPAIRING DIFFERENT CLASS OF BUGS WITH 50 TESTS.

Bug type	Const	Arith	Comp	Logic	Code Missing	Redundant Code	All
Total	14	14	16	10	27	9	90
SemFix	10	6	12	10	5	5	48
GenProg	3	0	5	3	3	2	16

TABLE VII
COREUTILS SUBJECT PROGRAMS

Subject Prog.	Failure Description	LOC
mknod	Segmentation fault.	183
mkdir	Segmentation fault.	159
mkfifo	Segmentation fault.	107
cp	Failed to copy a file (a “named pipe”).	2272

bugs. v) other reasons that we failed to identify (8 versions).

B. Experience with Coreutils

In this section, we present our experience in repairing GNU core utilities — Coreutils. In total, we were able to locate 9 buggy programs among the ones reported in earlier research [14]. Tests available in *Coreutils* 6.11 are utilized to encode the desired functionality of each utility. SEMFIX was able to repair 4 out of these 9 buggy programs. Table VII shows the 4 bugs that SEMFIX was able to repair. Three utilities *mkdir*, *mknod* and *mkfifo* have similar defects. In each utility, a null pointer access caused by wrong variable usage makes the program crash. SEMFIX found one variable as a replacement that let the program produce expected output. For *cp*, the condition guarding a file deleting operation is wrong. Because of the bug, the file deleting operation was not executed when it should be in the failing test. The status of the file is checked later. SEMFIX suggested the function checking the file status to return true indicating that the file is already deleted. For the remaining 5 buggy programs, SEMFIX could not find any 1-statement transformation to repair the buggy programs.

C. Comparison with GenProg on SIR and Coreutils

GenProg [6] is a genetic programming based repair tool. It uses mutation (i.e. *delete*, *insert*, *replace*) and crossover operators to search for a program variant that passes all tests. By applying these stochastic search operators, GenProg has been successful in repairing a variety of applications.

We compare the success rate (Table V) and running time (Fig. 4) of GenProg and SEMFIX. The test suites used for running GenProg are the same as the ones used in Section VI-A. We take the implementation of GenProg from http://dijkstra.cs.virginia.edu/genprog/. The configuration used to run GenProg is taken from [6] (i.e. $PopSize = 40$ and a maximum of 10 generations). The time bound for each repair trial of GenProg is set the same as that of SEMFIX, which is 4 minutes.

As shown in Table V, SEMFIX outperforms GenProg in all subject programs but Schedule2. Most of the bugs in Schedule2 are code missing bugs. Our tool could not find any 1-statement transformation to repair these bugs. With regard to running times, SEMFIX is strictly faster than GenProg (see

Figure 4). Since GenProg suffers from the burden of compiling and testing program variants, it takes longer for GenProg to find a repair when the size of test suite increases.

We also compare the results of SEMFIX and GenProg on different classes of bugs (Table VI). SEMFIX has a higher success rate than GenProg for each type of bugs. For code missing bugs, neither SEMFIX nor GenProg has a high success rate. We also notice that GenProg failed for all arithmetic bugs. This confirms that if the repair expression does not exist in other places of the program, genetic programming based approaches have rather low chance of synthesizing the repair.

We also employed GenProg to repair the bugs in Coreutils. Out of the 9 buggy programs, GenProg repaired the same number (4) of bugs as SEMFIX. Regarding the running time, GenProg took an average of 6 minutes while SEMXFIX took an average of only 3.8 minutes.

D. Comparison with Expression Enumeration

Our approach employs program synthesis to solve the repair constraint in order to find a successful repair. Program synthesis essentially searches among the set of possible expressions that can be constructed using the provided basic components and program variables. We conducted the following experiments to examine whether program synthesis is more favorable than explicit enumerating possible expressions.

The enumeration based program repair that we compare with proceeds as follows: i) It uses statistical debugging to generate a ranked list of repair candidates. ii) It scans through the result of statistical debugging. For each scanned repair candidate, it enumerates all possible expressions that can be constructed based on a given set of components and program variables. Thus, given variables $\{x, y, z\}$ and operator $+$, it tries out $\{x + y, x + z, y + z, x + x, y + y, z + z\}$ to see if any of them is a successful repair. The enumeration terminates until a successful repair is generated, which is confirmed by running all tests on the repaired program. We use the same set of components and variables as in our program synthesis step. Following the layers in Table III, we enumerate from simple expressions to more complex expressions. We have also tried to avoid redundant enumeration. Only one of $x + y$ and $y + x$ is tried out, and expressions like $x - x$ are not tried out at all.

One of the problem with enumeration is the handling of constants in expressions. Enumeration based method is unlikely to figure out the desired value of constant unless it falls within the commonly used ones, e.g. {-1, 0, 1}. Therefore, we use symbolic execution to figure out the value of constant when an enumerated expression contains unknown constant variables. Suppose we want to see whether any expression of the form `x+con` could be a valid repair, where x is a program variable and `con` is some unknown constant. We first replace the repair candidate with `x+con` and then mark `con` as symbolic and use the method in Section IV-A to generate a constraint over the symbolic value of `con`, solving which gives us a concrete value of `con`.

We use the same set of SIR programs as in Table IV in this comparison. For each buggy program version, the same set of 50 tests are used as in Section VI-A. The enumeration based program repair was successful for 45 out of 90 program versions, and all these 45 versions are contained in the 48 versions that SEMFIX was able to repair. For each of the 3 cases that SEMFIX succeeded but enumeration method failed, enumeration had to examine more than 9000 expressions and timed out after 20 minutes. In contrast, synthesis based repair took only 94 seconds on average for these 3 cases to produce a repair. As an example, in one version of TCAS, SEMFIX generated

```
tmp = ((Other_Capability<Alt_Layer_Value)?
Two_of_Three_Reports_Valid:Cur_Vertical_Sep);
```

as a replacement for `tmp = Up_Separation`. This is a non-trivial fix generated which the program synthesis allows SemFix to generate, and this could not be generated by the enumeration method even when it was augmented with symbolic execution to solve for constants.

TABLE VIII
AVERAGE RATIO OF RUNNING TIME OVER ALL VERSIONS OF A SUBJECT PROGRAM(TIME TAKEN BY ENUMERATION BASED REPAIR VS. TIME TAKEN BY SEMFIX).

Subject	Tcas	Schedule	Schedule2	Replace	Grep	**All**
Ratio	6.9	2.8	2.5	1.36	2.2	**4.16**

Table VIII compares the running time of SEMFIX and enumeration based repair. On average, the time taken for the enumeration method is 4.16 times of the time taken by SEMFIX.

VII. THREATS TO VALIDITY

a) Generalization of the fix: The repair generated by SEMFIX may not be generalized to test cases that are not in the given test suite. This is inherently due to the fact that our repair is guided by a given test suite, not a formal specification. To minimize the effort required from users, we choose to extract intended program semantics from a given test suite via symbolic execution.

b) Other statistical debugging metrics: The choice of statistical debugging metrics could potentially affect the effectiveness of our technique. On this front, we have also tested SEMFIX with another popular statistical debugging metric – Ochiai [18]. We found that using Ochiai instead of Tarantula has negligible impact on our experiment results. For example, using Ochiai instead of Tarantula, SEMFIX was able to fix two more versions of Tcas but one less version of Schedule.

VIII. RELATED WORK

Recently, there has been a growing interest in automated program repair. We have compared our technique with genetic programming [2], [6]. Several program repair approaches assume the existence of program specification. AutoFix-E [1] and AutoFix-E2 [19] are based on the program contracts in Eiffel programs. Fixes are generated following predefined schema so that the fixed programs satisfy the corresponding program contracts. Jobstmann, et. al. [20] uses LTL specifications for finite state programs. The process of finding a repair is reduced to a game, a winning strategy of which corresponds

to a successful repair. Gopinath, et. al. [3] use behavioral specifications and encode the specification constraint on the buggy program into SAT constraint, a solution of which leads to a repair. Fixes are only generated by replacing variables used in assignments. Robert and Roderick [21] employ template based repair for linear expressions. Template parameters are figured out through symbolic execution. In contrast to symbolic execution in our approach, their symbolic execution considers all program inputs and template parameters as symbolic. He and Gupta [22] compute weakest pre-condition along an execution trace to infer the desired program state. A repair is generated based on the difference between the actual program state during the failing execution and the desired program state.

Dallmeier, et. al. [23] try to generate fixes from object behavior anomalies. Normal program properties are mined from successful executions and fixes are generated so that previous failing executions can also satisfy the mined properties. ClearView [24] follows a similar scheme but works on deployed binary program when high availability is required. Learning from history repair data, BugFix [25] suggests bugfix that has been used in a similar debugging situation. The debugging situation captures both static and dynamic information to increase accuracy. Debroy and Wong [26] propose to use mutation for program repair. A number of mutated program versions are generated and tested in a trial-and-error fashion. PHPRepair [27] focuses on HTML generation errors in PHP programs. Constraints on string literals are collected from test executions and solved through string solver. The derived string literals from string constraint solving are then used to repair the PHP programs.

Instead of fixing a buggy program, program sketching [7], [8] allows a programmer to write a sketch of the implementation idea while leaving the low level details omitted as *holes* to be automatically filled up by the sketch compiler. The programmer is also required to provide the program specification in the form of a reference program. In contrast, SEMFIX requires neither the sketch nor the program specification. Instead, SEMFIX tries to synthesize a glimpse of the intended program specification through symbolic execution of the given test suite.

Our usage of symbolic execution is similar to that of angelic debugging [10]. For each expression in a buggy program, angelic debugging uses symbolic execution to check whether it is possible to modify the expression to fix the failing tests without breaking any passing tests.

IX. CONCLUSION

We propose SEMFIX as a semantics based program repair tool. SEMFIX derives *repair constraint* from a set of tests and solves the repair constraint to generate a valid repair. SEMFIX is able to synthesize a repair even if the repair code does not exist anywhere in the program. Our experimental results demonstrate that SEMFIX is able to fix various types of bugs and outperforms search based program repair technique.

ACKNOWLEDGMENTS

This work was partially supported by Singapore Ministry of Education grant MOE2010-T2-2-073.

REFERENCES

[1] 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*, 2010.
[2] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*, 2009.
[3] D. Gopinath, M. Malik, and S. Khurshid, "Specification-based program repair using SAT," in *TACAS*, 2011.
[4] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," in *OOPSLA*, 2003.
[5] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard, "Inference and enforcement of data structure consistency specifications," in *ISSTA*, 2006.
[6] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *ICSE*, 2012.
[7] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," in *PLDI*, 2005.
[8] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, 2006.
[9] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *PLDI*, 2003.
[10] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *ICSE*, 2011.
[11] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *ICSE*, 2010.
[12] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, 2005.
[13] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE*, 2002.
[14] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.
[15] S. Anand, C. Păsăreanu, and W. Visser, "JPF–SE: A symbolic execution extension to java pathfinder," in *TACAS*, 2007.
[16] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *PLDI*, 2011.
[17] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008.
[18] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *PRDC*, 2006.
[19] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer, "Code-based automated program fixing," in *ASE*, 2011.
[20] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *CAV*, 2005.
[21] R. Könighofer and R. Bloem, "Automated error localization and correction for imperative programs," in *FMCAD*, 2011.
[22] H. He and N. Gupta, "Automated debugging using path-based weakest preconditions," in *FASE*, 2004.
[23] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *ASE*, 2009.
[24] J. H. Perkins *et al.*, "Automatically patching errors in deployed software," in *SOSP*, 2009.
[25] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, "Bugfix: A learning-based tool to assist developers in fixing bugs," in *ICPC*, 2009.
[26] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *ICST*, 2010.
[27] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, "Automated repair of HTML generation errors in PHP applications using string constraint solving," in *ICSE*, 2012.