2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

DirectFix: Looking for Simple Program Repairs

Sergey Mechtaev, Jooyong Yi and Abhik Roychoudhury
National University of Singapore
{mechtaev,jooyong,abhik}@comp.nus.edu.sg

***Abstract*—Recent advances in program repair techniques have raised the possibility of patching bugs automatically. For an automatically generated patch to be accepted by developers, it should not only resolve the bug but also satisfy certain human-related factors including readability and comprehensibility. In this paper, we focus on the simplicity of patches (the size of changes). We present a novel semantics-based repair method that generates the simplest patch such that the program structure of the buggy program is maximally preserved. To take into account the simplicity of repairs in an efficient way (*i.e.*, without explicitly enumerating each repair candidate for each fault location), our method fuses fault localization and repair generation into one step. We do so by leveraging partial MaxSAT constraint solving and component-based program synthesis. We compare our prototype implementation, DirectFix, with the state-of-the-art semantics-based repair tool SemFix, that performs fault localization before repair generation. In our experiments with SIR programs and GNU Coreutils, DirectFix generates repairs that are simpler than those generated by SemFix. Since both DirectFix and SemFix are test-driven repair tools, they can introduce regressions for other tests which do not drive the repair. We found that DirectFix causes substantially less regression errors than SemFix.**

I. Introduction

Simple is better! When repairing a program, it is preferable to construct patches which are simple and readable. This is because responsible software maintainers would not blindly accept a suggested patch; rather, they would review and inspect a patch carefully before accepting it [1], [2] – which occurs only if they judge that the patch is correct (i.e., the bug is resolved) and safe (i.e., no new bug is induced). They would also modify the patch and add more tests, if necessary. Thus, simple and small patches would be more easily accepted by maintainers than more complex alternatives. The ease of acceptance, as well as abundance of small/simple patches are confirmed by the studies of [3], [4]. Hence it is instructive to have program repair tools produce small patches. To the best of our knowledge, existing automatic repair tools such as GenProg [5], SemFix [6] and PAR [7] do not explicitly take into account of the simplicity of a patch while generating patches, although more general issues about patch quality (e.g., patch maintainability [8] and users' willingness to accept patches [7]) have been raised and studied in recent years.

Finding a simple repair (we use "repair" and "patch" interchangeably) is not necessarily simple. In fact, it is challenging to find the simplest (or a simple enough) repair among many possible repairs, without enumerating each repair. Note that even for finding one repair, existing repair tools often take substantial amount of time. We propose in this paper an efficient test-driven repair method (and its implementation DirectFix) that can find simple repairs. Our repair method is test-driven as in GenProg, SemFix and many other existing repair methods. Our key observation is that the simplicity of a repair is influenced by the choice of the program location that is modified in a repair. If unsuitable program locations are chosen to be modified, the corresponding repair is also likely to be suboptimal (meaning unduly complex repairs). In the next section, we show examples of such unnecessarily complex repairs.

Existing test-driven repair methods rely on statistical fault localization [9]–[11] to choose program locations to modify (often called fault locations in the literature [12]). In general, fault locations are selected in proportion to their suspiciousness scores. High suspiciousness scores are assigned to the program locations that execute more frequently in failing tests. However, the simplicity of repairs is not a part of suspicious score equations, and thus these scores have no direct relationship with how simple a repair is. To include the simplicity of repairs into the logic of choosing fault locations, we perform fault localization and repair generation simultaneously in a combined manner.

The *main intuition* behind our approach is to (a) fuse the fault localization and repair steps into a single step via partial MaxSAT solving, (b) ensure that the resultant fused method still remains scalable by using the buggy program as a reference - we choose repairs which will cause minimal changes to the buggy program.

The *main technical contribution* of this paper is to integrate fault localization and repair generation in an efficient way – without explicitly enumerating each repair candidate for each fault location. We achieve this by reducing the problem of program repair into an instance of the Maximum Satisfiability problem (more specifically, a Partial MaxSMT problem). For a given buggy program and a test suite, we formulate a logical formula in a way that a model (satisfiable assignment) of this formula is the simplest repair – simplest in the sense that the structure of the original buggy program is preserved as much as possible. While the nature of MaxSMT allows removing existing expressions of a buggy program (our simple repairs are suggested at the expression level), we can replace those removed expressions with new ones by using component-based program synthesis [13]. We implement our approach into a tool, DirectFix, that formulates a necessary formula and solves it using our Partial MaxSMT solver implemented on top of Z3 SMT solver [14]. We also evaluate our tool on in total 98 buggy versions of SIR programs and 9 real bugs of GNU Coreutils, which exemplify the mistakes programmers can

```
1  x = E1; // E1 represents an expression.
2  y = E2; // E2 represents an expression.
3  S1; // S1 represents a statement. Neither x nor y is redefined by S1.
4  if (x > y) // FAULT: the conditional should be x >= y
5    return 0;
6  else
7    return 1;
```

(a) A buggy program snippet; a bug is in line 4.

```
1  x = E1; y = E2;
2  if (x == y) { S1; return 0; } // This line is one possible repair.
3  S1;
4  if (x > y)
5    return 0;
6  else
7    return 1;
```

(b) A repair that resembles a GenProg repair

```
1  x = E1;
2  y = E2;
3  S1;
4  if (x >= y) // SIMPLE FIX: >= is substituted for >
5    return 0;
6  else
7    return 1;
```

(c) An alternative simpler repair; an operator is replaced.

Fig. 1. The first motivating example

```
1  if (x > y)    // FAULT 1: the conditional should be x >= z
2    if (x > z) // FAULT 2: the conditional should be x >= y
3       out = 10;
4    else
5       out = 20;
6  else out = 30;
7  return out;
```

(a) A buggy program snippet; bugs are in line 1 and 2.

```
1  if (x > y)
2    if (x > z)
3       out = 10;
4    else
5       out = 20;
6  else out = 30;
7  return ((x>=z)? ((x>=y)? 10 : 20) : 30); // This line is one possible repair.
```

(b) A repair that resembles a SemFix repair

```
1  if (x >= z)    // SIMPLE FIX: >=z is substituted for >y
2    if (x >= y) // SIMPLE FIX: >=y is substituted for >z
3       out = 10;
4    else
5       out = 20;
6  else out = 30;
7  return out;
```

(c) An alternative simpler repair; operators and variables are replaced.

Fig. 2. The second motivating example

often make. Despite the limited size of our subject programs and tool limitations inherited from the underlying tools upon which DirectFix is built – most notably, VCC [15], which transforms a C program into a logical formula, currently cannot handle floating point arithmetic; in such cases, we designated the (transformable) suspicious functions, assuming that developers have insight about potential buggy functions –, the overall experimental results are promising. DirectFix suggests repairs successfully 59% of the time. Moreover, 56% of those repairs are equivalent to the ground truth repairs, and 89% of them alter the same program line(s) as the ground truth versions. Such figures are significantly higher than when SemFix [6] is applied to the same subjects with the same test suites and the same information about suspicious functions (i.e., more than 3 times of equivalent repairs and more than 2 times of same-line repairs). Recall that SemFix performs fault localization and repair as separate steps, and does not consider the simplicity of the repairs. We also found in our experiments that DirectFix repairs cause regression errors (when checked against the test universe and not just the test-suite driving the repair) less frequently than SemFix repairs.

II. Motivating Examples

We present three simple yet motivating examples in this section (in Section VII, we also present our repairs for actual programs). Consider the program snippet in Fig. 1(a). This program is supposed to return 0 if x >= y holds at the end of the program; otherwise, it should return 1. However, the developer of this program made a small mistake of not considering a case of x==y. Here, Fig. 1(b) and 1(c) show two different valid repairs. Notice that the former repair is more complicated than the latter one. Most developers would prefer the second simpler repair. To the best of our knowledge, existing repair tools do not take account of how simple a repair is. They stop looking for a repair once one is found, no matter how complex that repair is. Indeed, a repair in Fig. 1(b) resembles a repair generated by GenProg [5]. GenProg grafts existing code onto a buggy program in an attempt of repair. As a result, GenProg often generates repairs that look nonsensical to human developers, as pointed out in [7].

Meanwhile, a more recent repair tool, SemFix [6], seems to generate simpler repairs than GenProg (user-studies are yet to be conducted to fully validate this, but intuitively this is so because SemFix [6] performs repair at the expression level, unlike GenProg that performs repair at the statement level). However, SemFix still often generates repairs that are more complex than necessary. Fig. 2 shows such an example. Given a buggy program in Fig. 2(a) – the first two lines are mistakenly swapped, and the equal signs (=) are omitted –, SemFix can generate a repair shown in Fig. 2(b). Compare this repair with an alternative repair shown in Fig. 2(c). The latter repair is simpler despite that it modifies two lines of a program (SemFix cannot modify multiple lines). These two examples also show that a buggy program can be repaired in multiple ways producing repairs of varying simplicity.

There is one more important reason for selecting a repair carefully: the reliability of a repaired program (the likelihood that the repaired program not only resolves bugs in the given test-suite, but also does not introduce new bugs shown by tests outside the test-suite) varies depending on a selected repair. Consider a buggy program in Fig. 3(a) that checks whether the character c is included in the string (character array) s. The table in Fig. 3(b) shows the expected and actual input/output relationship. The first test fails because all the characters of string s are not scanned while looking for the same character as the one in c. Notice in the table that variable k does not

```
// FAULT: k is NOT equal to the length of array s.
for (i=0; i<k; i++)
  if (s[i] == c) return TRUE;
return FALSE;
```

(a) A buggy program that checks if the character c is included in string s.

Input			Output	
s	c	k	expected	actual
"ab?"	'?'	2	TRUE	FALSE
"ab?c"	'?'	3	TRUE	TRUE
"!ab"	'!'	2	TRUE	TRUE

(b) Expected input and output

```
for (i=0; i<k; i++)
  // The following line is one possible repair.
  if (c == '?' || c == '!') return TRUE;
return FALSE;
```

(c) A (buggy) repair that passes the above tests

```
for (i=0; i<=k; i++) // SIMPLE FIX: <= is substituted for <
  if (s[i] == c) return TRUE;
return FALSE;
```

(d) A more reliable repair

Fig. 3. The third motivating example

hold the value of the length of s, it holds a value one less than the length. As before, more than one repair exist for this buggy program. Fig. 3(c) and 3(d) show two possible repairs – both repairs pass all the tests in Fig. 3(b). However, the first repair (Fig. 3(c)) looks hazardous. What if a character other than '?' or '!' is searched for? While such potential hazard of a repair can be diminished by choosing a right test suite, what is a right test suite is another important research question that has not been thoroughly addressed yet.

Meanwhile, the second simpler repair (Fig. 3(d)) preserves the original correct behavior, as well as correcting the buggy behavior. The contrast between these two repairs suggests the following hypothesis. The rationale behind the hypothesis is that simpler repairs are likely to modify the behavior of a program in a more restricted fashion.

Hypothesis. *Simple repairs are less likely to change the correct behavior of the original version than more complex repairs. Thus, simple repairs are likely to be less hazardous.*

Existing test-driven program repair tools perform fault localization upfront, and search for a repair around the program locations marked suspicious at the fault localization phase. Therefore, a straightforward way to find the simplest repair is to iteratively generate a repair at each combination of suspicious program locations, and select the simplest repair. However, it is apparent that this straightforward approach would not scale, considering the fact that even finding a single repair often takes substantial amount of time. To find simple repairs more efficiently (without explicitly enumerating each repair candidate), we integrate the two phases of program repair – (i) fault localization and (ii) repair search – into a single step.

III. Background

DirectFix is a semantics-based program repair approach that exploits recent advances of SMT solvers. It reduces repair problem to Maximum Satisfiability problem. Particularly, this approach constructs a logical formula, a solution to which corresponds to a fix. Our encoding is based on Component-based Synthesis [13] extended to produce syntactically minimal changes as well as to improve scalability.

A. Preliminaries

The *Satisfiability problem* in propositional logic (SAT) is the problem of determining whether a given formula φ has a model. *Maximum Satisfiability* (MaxSAT) is a generalization of SAT whose goal is to find the maximum number of clauses of a given formula that can be satisfied. *Satisfiability Modulo Theories* (SMT) is a satisfiability problem with respect to given background theories. MaxSMT is a generalization of MaxSAT on SMT. *Partial Maximum Satisfiability* (pMaxSAT) for a set of soft clause s and a set of hard clauses h is a problem of finding the maximum subset s_{max} of clauses s such that $s_{max} \wedge h$ is satisfiable. pMaxSMT is a generalization of pMaxSAT on SMT.

DirectFix utilizes program semantics expressed through a logical formula called *trace formula* in the literature [16], [17].

Definition 1 (Trace Formula)**.** *A trace formula TF of a deterministic program P is a logical formula that satisfies the following property. Given the input I of program P,*

$$TF(I,O) \text{ is } \begin{cases} \textit{satisfiable if the output of } P \textit{ is } O \\ \textit{unsatisfiable if the output of } P \textit{ is not } O \end{cases}$$

where $TF(I,O)$ denotes the trace formula TF whose input variable is bound with I, and output variable with O.

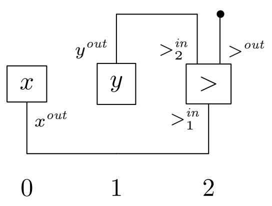
B. Component-based Synthesis [13]

Expressions and, in general, programs that satisfy given requirements can be generated automatically through *Component-based Synthesis* [13] (CBS). Here we present the original CBS technique, our extensions and optimizations are given in Section V.

Definition 2. *Let v be a variable, V be a set of variables such that $v \notin V$, F be a set of operators, $\mathcal{O}$ be a constraint over $\{v\} \cup V$ called oracle. Component-based Synthesis (v, V, F, $\mathcal{O}$) is a problem of finding an expression e such that*

- *e is constructed using a subset components $C = V \cup F$ and constants;*
- *$\mathcal{O} \wedge (v = e)$ is satisfiable.*

This approach constructs expressions by connecting primitive building blocks called *components* with each other. Components could be constants or variables or operators. The correct linkage between components is determined using an SMT solver. Specifically, semantics of components, semantics of the connections between them as well as the oracle constraint are used to construct an SMT formula which we call *Component-based Encoding* (CBE).



$$L(>^{out}) = 2$$
$$L(>^{in}_1) = 0$$
$$L(>^{in}_2) = 1$$
$$L(x^{out}) = 0$$
$$L(y^{out}) = 1$$
$$L(v^{in}) = 2$$

Fig. 4. The inputs and output of the components of expression $x > y$ are allocated in the interval $[0, 3)$. The output is bound by the variable v.

The idea of CBE is to consider expressions as circuits. Each component has a set of inputs and an output. Then, to synthesize desired expression, the solver needs to find the right connections between components inputs and outputs. In order to capture information about connections, for each input and output, a numeric variable called *location variable* is defined. The meaning of location is straightforward: an input and an output are connected iff they have the same location.

CBE defines relationship between values and locations of components' inputs and outputs. For each input and output, we introduce a variable that corresponds to its value. We use the following notation: for component c, the variable c^{out} corresponds to the value of its output, the variable c^{in}_k is the value of its k-th input. We indicate the number of inputs of c as $NI(c)$. We denote location variables using the function L. For example, the location variable of the first input of the component c is $L(c^{in}_1)$. Fig. 4 demonstrates an example of an assignment of locations variables. The components of the expression $x > y$ are allocated within the interval $[0, 3)$. The output of the variable x has location 0, the output of the variable y has location 1, and the output of the operator $>$ is located at 2. The first input of $>$ is linked to x, the second input of $>$ is linked to y. Assume that the output of the expression is bound by the variable v, that is $v = x > y$. The output of the circuit is marked by the bullet and is indicated by the constraint $L(v^{in}) = 2$. The expression $x > y$ can be trivially reconstructed using the values of the location variables. We assume that there is a function `Lval2Prog` that builds an expression from an assignment of location variables.

CBE consists of three types of constraints: *well-formedness constraints*, *semantics constraints* and *connections constraints*. Well-formedness constraints (ϕ_{wpf}) restrict location variables so that any satisfying assignment to these variables corresponds to an expression of a valid structure. These constraints include range constraints (ϕ_{range}) that allocate all components inputs and outputs within a range, consistency constraints (ϕ_{cons}) that ensure that the output of each component has unique location, and acyclicity constraints (ϕ_{acyc}) that forbid cyclic connections.

$$\phi_{\text{wpf}} \overset{\text{def}}{=} \phi_{\text{range}} \wedge \phi_{\text{cons}} \wedge \phi_{\text{acyc}}$$

$$\phi_{\text{range}} \overset{\text{def}}{=} \bigwedge_{c \in \{v\} \cup C} \Big(0 \le L(c^{out}) < |C| \wedge \bigwedge_{k \in [1..NI(c)]} 0 \le L(c^{in}_k) < |C|\Big)$$

$$\phi_{\text{cons}} \overset{\text{def}}{=} \bigwedge_{(c,s) \in C \times C, c \neq s} L(c^{out}) \neq L(s^{out})$$

$$\phi_{\text{acyc}} \overset{\text{def}}{=} \bigwedge_{c \in C, k \in [1..NI(c)]} L(c^{out}) > L(c^{in}_k)$$

Semantics constraints (ϕ_{lib}) are defined for each component. They specify the semantics of the component as the relations between its inputs and output. For example, semantics constraint for the component c corresponding to the addition operation is defined as $c^{out} = c^{in}_1 + c^{in}_2$. Semantics constraints for a component c corresponding to a variable x is defined as $c^{out} = x$. Similarly, the semantics constraint for a component c corresponding to a constant a is $c^{out} = a$.

Connections constraints (ϕ_{conn}) capture the semantics of an expression to be synthesized through the semantics of the components and the connections between them.

$$\phi_{\text{conn}} \overset{\text{def}}{=} \bigwedge_{\substack{(c,s) \in C \times \{v\} \cup C \\ k \in [1..NI(s)]}} L(c^{out}) = L(s^{in}_k) \Rightarrow c^{out} = s^{in}_k$$

Theorem 1. *Let v be a variable, V be a set of variables such that $v \notin V$, F be a set of integer operators, $\mathcal{O}$ is a constraint over $\{v\} \cup V$. $\phi = \phi_{wpf} \wedge \phi_{lib} \wedge \phi_{conn} \wedge \mathcal{O}$ is a Component-based Encoding, that is* `Lval2Prog` *produces a solution to the Component-based Synthesis problem (v, V, F, $\mathcal{O}$) taking as input v, $C = V \cup F$ and any model of ϕ as an assignment of the location variables.*

Described CBE is only suitable for representing oracle corresponding to a single test case. Indeed, if we conjoin input-output constraints for different test cases, the formula is trivially unsatisfiable. To extend this encoding for several test cases, we rename variables in the encoding formula so that each test case uses unique variables names. Then, the formula that captures all given input-output relationships is a conjunction of renamed formulas for each test case.

IV. Overview of our Approach

To find a repair, we first translate a given buggy program into a trace formula. For example, Fig. 5(b) demonstrates the trace formula for the function foo shown in Fig. 5(a). This function is buggy, and its test test_foo fails (we use a single test in this example for simplicity). The given test is translated into the following oracle constraint:

$$\mathcal{O} \overset{\text{def}}{=} (x_1 = 0) \wedge (y_1 = 0) \wedge (result = 3)$$

The conjunction $\varphi_{buggy} \wedge \mathcal{O}$ is unsatisfiable, reflecting the fact that the test fails.[1]

Our goal is to find which expressions of φ_{buggy} need to be modified and how they should be modified, so that this modified formula φ_{repair} makes $\varphi_{repair} \wedge \mathcal{O}$ satisfiable. In our example, the ground truth repair is as follows:

[1]If there are multiple tests, say two, we formulate $Rename(\varphi_{buggy} \wedge \mathcal{O}_1) \wedge Rename(\varphi_{buggy} \wedge \mathcal{O}_2)$, where function $Rename$ returns the input formula after replacing its variables with fresh variables.

```
int foo(int x, int y) {
  if (x > y)  // FAULT: the conditional should be x >= y
    y = y + 1;
  else
    y = y − 1;
  return y + 2;
}

void test_foo() { assert (foo(0,0)==3); }
```

(a) A buggy function and its test

$$\varphi_{buggy} \equiv (\text{if } (x_1 > y_1) \text{ then } (y_2 = y_1 + 1) \text{ else } (y_2 = y_1 - 1)) \\ \wedge (result = y_2 + 2)$$

(b) The trace formula φ_{buggy} for foo; variables x_i and y_i correspond to the program variables x and y, respectively, and $result$ to the return value of the program

Fig. 5. A trace formula is constructed from a buggy program and its tests

$$\varphi_{repair} \overset{\text{def}}{=} (\text{if } (x_1 \geq y_1) \text{ then } (y_2 = y_1 + 1) \text{ else } (y_2 = y_1 - 1)) \\ \wedge (result = y_2 + 2)$$

Essentially, our repair method views a program as a circuit. To generate a fix, it (i) cuts some of the existing connections and (ii) adds new components and connections. To obtain the simplest (the least destructive) repair, we want to cut as few connections as possible. We achieve this by reducing the problem of program repair into an instance of the maximum satisfiability (MaxSAT) problem – more specifically, a partial MaxSMT (pMaxSMT) problem.

To generate a repair based on pMaxSMT, we construct a formula that we call *repair condition*. Given a trace formula φ_{buggy}, the repair condition φ_{rc} is the following:

$$\varphi_{rc} \overset{\text{def}}{=} (\text{if } v_1 \text{ then } (y_2 = v_2) \text{ else } (y_2 = v_3)) \wedge (result = v_4) \\ \wedge cmpnt(v_1 = x_1 > y_1) \wedge cmpnt(v_2 = y_1 + 1) \\ \wedge cmpnt(v_3 = y_1 - 1) \wedge cmpnt(v_4 = y_2 + 2)$$

The above formula φ_{rc} is semantically identical with φ_{buggy}. The only difference is that we substitute fresh variables v_i for the rvalue expressions of φ_{buggy}, while keeping the equality relationship between each v_i and the expression it represents (e.g., $v_1 = x_1 > y_1$) inside the $cmpnt$ function. This function componentizes its parameter expression into a circuit form, following the idea of Component-based Synthesis.

To obtain the simplest (the least destructive) repair, we use a pMaxSMT solver. In pMaxSMT, a formula is split into (i) hard clauses (clauses that must be satisfied) and (ii) soft clauses (clauses that do not have to be satisfied). In hard clauses, we include the clauses that express the semantics of the component and the oracle data. Meanwhile, with soft clauses, we constrain the structure of the program expressions. For example, we construct the *structure constraint* for the expression $x > y$ as follows, which is essentially equivalent to the circuit diagram in Fig. 4:

$$L(>_1^{in}) = L(x^{out}) \wedge L(>_2^{in}) = L(y^{out}) \wedge L(>^{out}) = L(v^{in})$$

In the above constraint, we bind the output of the expression with a fresh variable v, that is $v = x > y$. As shown, this constraint specifies the connections between the components of the expression as well as its output binding. After splitting $\varphi_{rc} \wedge \mathcal{O}$ into hard clauses and soft clauses as described above, we feed $\varphi_{rc} \wedge \mathcal{O}$ into a pMaxSMT solver. Then, the solver removes some structure constraints (if necessary), and returns a model corresponding to a fix.

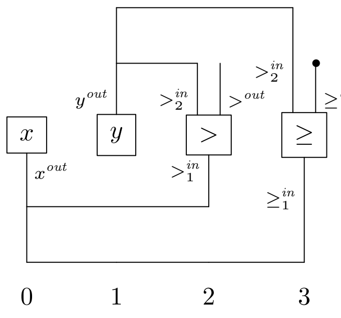


Fig. 6. Repairing expression $x > y$ by replacing $>$ with $\geq$.

Fig. 6 shows how a solver can modify the expression $x > y$ using an additional component $\geq$ in order to repair the program. Specifically, it removes one connection between the outputs of $>$ and the input of v corresponding to the structure constraint $L(>^{out}) = L(v^{in})$, and adds three new connections: (i) between the output of x and the first input of $\geq$, (ii) between the output of y and the second input of $\geq$, and (iii) between the output of $\geq$ and the binding variable v. Such new connections are obtained by using a model for the repair condition, namely, the values of the location variables.

We note that by looking for a model that maximizes the number of satisfied clauses of $\varphi_{rc} \wedge \mathcal{O}$, we effectively cut and add connections simultaneously. In other words, we perform fault localization and repair generation at the same time.

V. Methodology

While automated program repair has been shown to be effective, automatically generated patches can damage the structure of the original program and introduce regressions. To address this problem, we devise an approach that searches for syntactically minimal fixes. Our approach combines fault localization and correction into a single step, which is achieved by reducing repair problem to Partial Maximum Satisfiability Problem. Our pMaxSMT encoding is based on Component-based Synthesis encoding extended to capture the structure of the original program as well as to tackle scalability problems.

A. Repair Problem and Repair Condition

Unlike in CBS, our goal is to modify the existing expressions of a buggy program in a way that changed expressions make all tests pass. For this reason, we modify the synthesis problem (Definition 2) into the following repair problem.

Definition 3 (Repair Problem)**.** *Let v be a variable, V be a set of variables such that $v \notin V$, F be a set of integer operators, $\mathcal{O}$ be a constraint over $\{v\} \cup V$ called oracle. Let e be a possibly faulty expression constructed using a subset of components $C = V \cup F$ and constants such that $\mathcal{O} \wedge (v = e)$ is* *not satisfiable. Repair problem* $(v, e, V, F, \mathcal{O})$ *is a problem of finding a repaired expression* e' *such that*

- e' *is constructed using a subset of components* $C = V \cup F$ *and constants.*
- $\mathcal{O} \wedge (v = e')$ *is satisfiable.*

To solve a repair problem, we construct a logical formula which we call *repair condition* that consists of two groups of clauses: hard clauses and soft clauses. Algorithm 1 describes how we generate a repair condition, given a test suite TS and a trace formula TF as input. Our algorithm substitutes fresh variables v_i for the rvalue expressions (the expressions of TF that are originated from the conditionals or right-hand-side expressions of a given buggy program[2]) to construct the formula $TF[e_i \mapsto v_i]$. The formula $TF[e_i \mapsto v_i] \wedge (\bigwedge_i v_i = e_i)$ is semantically equivalent to the initial trace formula TF. However, expressions e_i that we allow to modify are now distinguished from the rest of the formula. Algorithm 1 applies CBE to all the components of $x_i = e_i$ together with additional component, and the formula $(\bigwedge_i \mathit{CBE}(v_i = e_i)) \wedge TF(I, O)[e_i \mapsto v_i]$ is returned as hard clauses of the repair condition for each test case $(I, O) \in TS$.

Meanwhile, we also extract the *structure constraint* ϕ_{struct} of each binding $v_i = e_i$, and classify ϕ_{struct} as a soft clause. The structure constraint of $v_i = e_i$ encodes the structure of expression e_i using location variables. It also encodes the binding of e_i to v_i using location variables. In the previous section, we showed that expression $x > y$ is encoded into the following structure constraint ϕ_{struct}:

$$\phi_{\text{struct}} \overset{\text{def}}{=} L(>_1^{in}) = L(x^{out}) \wedge L(>_2^{in}) = L(y^{out}) \wedge L(>^{out}) = L(v^{in}),$$

where v is a fresh binding variable. The structure constraint is obtained via the inverse function of `Lval2Prog` (`Lval2Prog` is a bijective function [13]).

Once a repair condition is obtained through Algorithm 1, we feed this repair condition to a pMaxSMT solver. If the solver finds a model, this model can be used to construct a repair expression using `Lval2Prog` introduced in Section III. Note that a pMaxSMT solver preserves as many original connections as possible, which guarantees that DirectFix changes the minimal number of program expressions, as formally described below.

Definition 4 (Simplicity of repair)**.** *Let* P *be a program,* TS *be a test suite with at least one failing test case,* e_i *be a subset of the expressions of* P*,* C *be a set of components. We call* P' *a* simple *repair of* P *if*

- P' *passes* TS*;*
- P' *can be obtained from* P *by substituting some of the subexpressions of* e_i *with expressions constructed from the components* C*;*
- *there is no program that passes* TS *and can be obtained from* P *using a smaller number of such substitutions.*

[2]The expressions of our TF are annotated with source code locations.

Algorithm 1 Repair condition generation

```
Input: trace formula $TF$ and test suite $TS$
Output: repair condition as a pair of hard and soft constraints
 1: $Hard, Soft \leftarrow True, True$ // Hard and soft clauses
 2: $Expr \leftarrow \{e \mid e$ is a rvalue expression of $TF\}$
 3: for test case $(I, O) \in TS$ do
 4:     for $e \in Expr$ do
 5:         $v \leftarrow$ a fresh variable
 6:         $C' \leftarrow$ select additional component for $e$
 7:         $TF(I, O) \leftarrow TF(I, O)[e \mapsto v]$ // replace $e$ with $v$
 8:         $\phi \leftarrow$ CBE for components of $e$ and components $C'$
 9:         $\phi_{\text{struct}} \leftarrow$ structural constraints for $v = e$
10:         $Hard \leftarrow Hard \wedge \phi$
11:         $Soft \leftarrow Soft \wedge \phi_{\text{struct}}$
12:     end for
13:     // To be able to bound a variable with a different value
14:     // in each test, we call function RENAME.
15:     $Hard \leftarrow$ RENAME$(Hard \wedge TF(I, O))$
16:     $Soft \leftarrow$ RENAME$(Soft)$
17: end for
18: return $Hard, Soft$
```

B. Optimization

The use of soft constraints reduces synthesis time. Our experiments demonstrate that a pMaxSMT solver implemented on top of an SMT solver can find a solution for a formula with soft constraints for some of the considered benchmarks, while the SMT solver for the same formula without soft constraints does not terminate within the timeout for all the benchmarks. This fact suggests that the use of the structure of the previous (buggy) versions improves synthesis performance.

For repairing some bugs, it is not sufficient to use only components that are already present in the buggy expressions. For such cases, we select *additional components* for each expression in the program. Selecting many additional components makes this approach not scalable. To address this limitation, we devise optimizations and heuristics that reduce the negative effect of additional components.

1) Sharing components: Selecting additional components for program expressions can significantly increase the search space for repair, which harms the scalability of the approach. For instance, if there are 10 program expressions and 10 program variables that we consider as additional components, then selecting each variable for each expression yields 100 variants to choose a single variable for repairing one of the expressions. However, proceeding on the assumption that the program is correct with the exception of a small part, we do not consider each component for each of the program expressions. Instead, additional components can be *shared* by several expressions. For instance, a variable can be shared by all the expressions from its scope.

The original CBE does not allow to share components between several expressions; in the original CBE, each expression has a fixed interval for allocating components and, consequently, a fixed set of available components. To alleviate this limitation, we extend CBE so that components of all the expressions are allocated in one big interval consisting of floating subintervals for each expression.

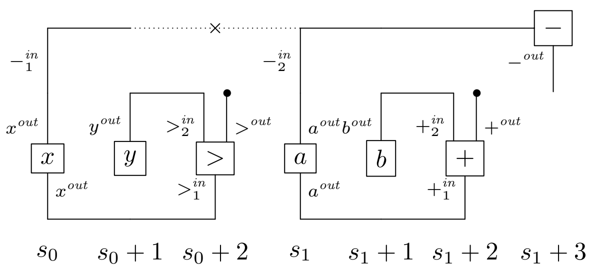


Fig. 7. Allocating components of the expressions $x > y$ and $a + b$ and the additional component "$-$" on the same interval using floating separators. Cross-expression connections are forbidden.

Allocating component for all the program expressions in one big interval requires introducing additional constraints to prevent invalid connections between component of different expressions. For this, we introduce a set of *separator variables* $\{s_i\}$ that define subintervals for each expression. Specifically, all the components of the expression e_j and the connections between them are allowed only within the interval $[s_{j-1}, s_j)$. Fig. 7 shows how the expressions $x > y$ and $a + b$ and the additional component "$-$" can be placed using such encoding. Note that the intervals for each expression are not fixed and can be extended to add the component "$-$". At the same time, we forbid the connections of the component "$-$" to cross the separator between $x > y$ and $a + b$ to prevent our tool from generating expressions of invalid structure.

Assume that we generate encoding for a set of program expressions $\{e_i\}$ for $i \in [1..N]$. The following constraints ensure that only valid connections are permitted:

$$\phi_{\text{range}} \stackrel{\text{def}}{=} \bigwedge_{\substack{c \in C \\ (i,j)=scope(c) \\ k \in [1..NI(c)]}} (s_i \le L(c^{out}) < s_j \wedge s_i \le L(c_k^{in}) < s_j)$$
$$\wedge \bigwedge_{i<m<j} \Bigg((s_m \le L(c^{out}) \wedge \bigwedge_{k \in [1..NI(c)]} s_m \le L(c_k^{in})) \vee (L(c^{out}) < s_m \wedge \bigwedge_{k \in [1..NI(c)]} L(c_k^{in}) < s_m) \Bigg)$$

where function *scope* maps a component c to an interval representing the range of program expressions where c can be used for repair. The first line of this formula specifies that each component is allocated within the intervals of the expressions from its scope. The second and third lines ensure that for each separator, the inputs and the output of each component are all placed either to the right of this separator or to the left, implying that connections do not cross the borders between expressions. ϕ_{cons}, ϕ_{acyc} and ϕ_{conn} are defined in an analogous manner to CBE, taking account of components' scopes.

Apart from components' constraints, we enforce the *interval consistency* constraint ϕ_{intcons} over separator variables to ensure that the interval for each expression is well-defined:

$$s_0 = 0 \wedge s_N = |C| \wedge \bigwedge_{[(i,j) \ | \ i,j \in [0..N], \ i<j]} s_i < s_j$$

where C is the set of all available components.

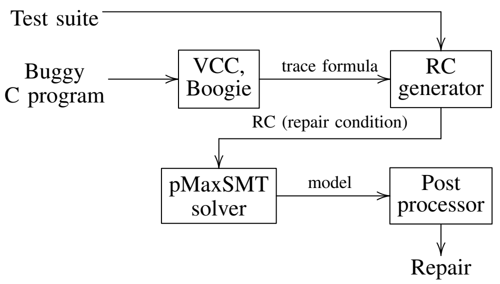


Fig. 8. The workflow of DirectFix

2) Typed-based space reduction: If the program to be corrected is statically typed, it is possible to use type information to reduce the search space for repair. We implement heuristics for the repair encoding that reduce the number of possible connections, the number of components and the number of candidate repair locations. In order to ensure that only well-typed expression are considered for repair, we modify the connection constraints so that inputs can be connected only with outputs of the same type.

Selecting a large number of additional components for repair yields considerable performance reduction. For this reason, we group component by their types into several *levels*: constants, boolean operators, arithmetical operators, comparison operators and variables. For each level, we generate and solve a separate repair condition. Grouping additional components by type allows us to utilize the following two heuristics. Firstly, we can prune program expression that cannot be repaired using additional components due to their type. For example, the statement $v = a \vee b$ cannot be repaired using integer arithmetics components. Secondly, we can reduce the number of connections between components in the original program expressions. Specifically, we do not split an expression into subexpressions if these subexpressions have incompatible type with the additional components and consider whole expression as one compound component. For example, the expression $a \vee x > y$ can be split into components a, $\vee$ and $x > y$ if we consider only boolean operators.

C. Handling Loops

For a loop, we unroll it k times; our trace formula guarantees that there is no execution paths requiring more than k unrolling. The consequence of loop unrolling is that the trace formula includes multiple instances of the program expressions that are executed several times inside loops. In order to make it possible to apply the fix generated by our tool to the original program, we need to ensure that all these expressions are modified *synchronously*. This is achieved by binding components' locations of these expression through auxiliary components called *phantom components*. Phantom components do not have semantics and are used only for binding location variables.

VI. Implementation

We implement the repair methodology described earlier into a prototype tool, DirectFix. The overall workflow of

TABLE I
SUBJECT PROGRAMS

Subject	LOC	#Versions	Description
Tcas	135	41	Air traffic control program
Replace	518	30	Text processor
Schedule	304	9	Process scheduler
Schedule2	262	9	Process scheduler
Coreutils	107 – 2909	9	Collection of OS utilities

DirectFix is shown in Fig. 8. To obtain a trace formula from a buggy program, we use two third-party tools, VCC [15] and Boogie [18]. VCC translates a C program into a Boogie program. Subsequently, the Boogie verifier takes as input a Boogie program, and generates a verification condition – a formula used to prove the absence of an error. Both VCC and Boogie can handle pointer arithmetics.

A verification condition generated from Boogie is very similar to a trace formula we need. The following shows the verification condition φ_{vc} of function foo we earlier showed in Fig. 5(a):

$$\neg((\text{if } (x_1 > y_1) \text{ then } (y_2 = y_1 + 1) \text{ else } (y_2 = y_1 - 1)) \\ \Rightarrow (result = y_2 + 2))$$

Notice that the trace formula φ_{buggy} we showed in Fig. 5(b) can be obtained by negating φ_{vc} and replacing $\Rightarrow$ with $\wedge$. Due to these subtle differences, $\varphi_{buggy} \wedge \psi_{test}$ is unsatisfiable as needed, while $\varphi_{vc} \wedge \psi_{test}$ is satisfiable. We modified the Boogie verifier in order to obtain a trace formula instead of a verification condition. Our trace formula is in SMT-LIB2 format [19] annotated with source code locations.

The trace formula of a buggy program and its test suite are fed into the RC (repair condition) generator of DirectFix, which is an implementation of Algorithm 1. Subsequently, the generated repair condition is fed into our pMaxSMT (Partial Maximum Satisfiability) solver we implemented on top of Z3 [14]. Our pMaxSMT implementation is the unsat-core-guided algorithm of Fu and Malik [20].

Finally, a model (satisfiable assignment) found by our pMaxSMT solver is post-processed to construct a patch. Currently, DirectFix shows which expressions are modified and how they are modified.

VII. EXPERIMENTAL RESULTS

In this section we present the experimental evaluation of DirectFix. We also compare the repairs generated by DirectFix with those generated by SemFix [6]. We ran our experiments on Intel Core i7-2600 CPU with Ubuntu 12.04 64-bit operating system. Table I shows our subject programs comprised of eighty nine buggy versions of four subject programs from SIR (Software-artifact Infrastructure Repository) [21] (the number of versions for each subject is shown in the #Versions column) and nine buggy versions of Coreutils reported by Cadar et al [22]. These subjects are the same as the ones used in the SemFix paper [6]. All our subjects come also with their correct versions, and we compare each of our repairs with its correct versions, if a repair is found. The timeout used in our experiments is 10^6 milliseconds (16 minutes and 40 seconds).

TABLE II
EXPERIMENTAL RESULTS

Subject	Repairs			
	Total	Equivalent (E)	Same Loc (S)	Diff (D)
Tcas	36 (87%)	19 (54%)	33 (91%)	2.28
Replace	11 (37%)	9 (81%)	10 (91%)	2.54
Schedule	4 (44%)	4 (100%)	4 (100%)	2.5
Schedule2	2 (22%)	1 (50%)	2 (100%)	2
Coreutils	5 (56%)	0 (0%)	3 (60%)	2
Overall	59%	56%	89%	2.26

For the subjects larger than Tcas, we designated the suspicious functions to reduce the search scope, assuming that developers have insight about which function might be buggy; for example, if a test fails after creating or modifying a function foo, then a bug is probably located in foo or its callees. For a library function whose source code is not available, we provided a model for it.

Table II shows the results of our experiments. Overall, 59% of buggy versions are repaired by DirectFix. More interestingly, 56% of those repairs are equivalent to the code in the correct versions. We take a repaired version as equivalent to its correct version when (i) the same program location is altered by the repair, and (ii) that alternative repair expression is logically equivalently to the corresponding expression in the correct version. Note that some expressions (e.g., x > 0 and x >= 1) are logically equivalent to each other, even though they are not syntactically identical.

Table II also exhibits that 89% of the repairs suggested by DirectFix alter the same program locations as those that differ from the correct versions (Equivalent repairs mentioned above are included in this category by definition). For example, DirectFix can suggest a new magic number instead of a buggy constant used in a buggy version. Although it is difficult to suggest the "correct" magic number in the absence of formal specification, the finding that simple replacement of a constant have all tests passing can be a good hint about where a bug is and what a repair should be.

As intended, our repairs are simple in most cases. To measure how simple our repairs are, we compare the original buggy version and a repaired version, and see how much they differ. More specifically, we compare the ASTs (Abstract Syntax Trees) of those two versions, and count (i) the number of AST nodes that are deleted from the buggy version and (ii) the number of AST nodes that are added into the repaired version. For example, if a buggy expression x > y is repaired into x >= y, then the counted number is two, because operator > is deleted from the buggy version, and >= is inserted into the repaired version.

TABLE III
COMPARISON WITH SEMFIX; E STANDS FOR EQUIVALENT, S STANDS FOR SAME LOC, D STANDS FOR DIFF, AND R STANDS FOR REGRESSION

Subject	Total	DirectFix				SemFix			
		E	S	D	R	E	S	D	R
Tcas	30	16	29	2.26	12	3	11	4.1	17
Replace	5	5	5	2.8	0	3	4	10.2	2
Schedule	4	2	4	2.5	1	1	4	8.5	3
Schedule2	2	1	2	2	1	1	2	5	2
Coreutils	4	0	3	2	-	0	0	4	-
Overall	44	53%	95%	2.31	31%	17%	46%	6.36	54%

The Diff column of Table II shows how much original buggy versions and their repaired versions differ in terms of AST differences described earlier. Overall, the differences between two versions are as low as 2.26, which is close to the optimal number 2 (the optimal number cannot be obtained sometimes when even the simplest repair requires changes of a few lines of a program or complicated changes).

The majority (56%) of our repairs are equivalent to ground truth repairs, and about 90% of our repairs alter the same program locations as ground truth repairs alter.

a) Quantitative Comparison with SemFix: We compare our repairs with those of SemFix [6]. Similar to DirectFix, SemFix also searches for repairs by analyzing the logical semantics of a program, and uses component-based synthesis to generate repairs. Further comparison between DirectFix and SemFix is given in Section IX. The core difference between SemFix and DirectFix is that DirectFix can search for simple conservative repairs whereas SemFix does not consider the simplicity of repairs. Thus, the comparison with SemFix is a good indicator about how effective our new approach is in terms of finding simple conservative repairs. We ran SemFix for the same subjects with the same tests as used for the DirectFix experiment. We also provided the same information about suspicious functions, so that only those suspicious functions and their callees can be modified. Table III compares the repairs that could be generated by both DirectFix and SemFix. Overall, the rates of equivalent repairs and same-location repairs are significantly higher in DirectFix than in SemFix (53% vs 17% and 95% vs 46%, respectively). Also, DirectFix repairs are simpler (less complex) than SemFix repairs as shown with Diff numbers (2.31 vs 6.36). We also compare how frequently regression errors are observed between DirectFix and SemFix. This is to test our hypothesis that simpler repairs are more likely to be safer. To observe regression errors, we apply the entire tests of our SIR subjects to repaired versions. SIR subjects have a huge number of tests, and we use no more than 50 tests to generate repairs in our experiments. We classify that there is a regression error if a repaired version produces a different output from the correct version in one of the entire tests. As shown in column R of Table III, regression errors are observed less frequently in DirectFix repairs than in SemFix repairs (31% vs 54%). This results coincides with the high rate of equivalent repairs of DirectFix – equivalent repairs by definition do not cause a regression error. However, DirectFix is slower than SemFix. For Tcas, for which we do not designate suspicious functions, DirectFix took an average of 3 minutes 20 seconds, while SemFix took an average of 9 seconds. For other benchmarks subjects where only specific functions are allowed to be modified, we perform repair on the unit level by reducing programs to only these functions as well as their dependencies. These reduced programs were given to both tools for fair comparison, after which both DirectFix and SemFix took less than a minute.

```
bool Non_Crossing_Biased_Climb() {
  int upward_preferred;
  int upward_crossing_situation;
  bool result;
  upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
  if (upward_preferred)
    result = !(Own_Below_Threat()) || ((Own_Below_Threat()) &&
              (!(Down_Separation >= ALIM())));
  else
    result = Own_Above_Threat() &&
            (Cur_Vertical_Sep >= MINSEP) && (Up_Separation >= ALIM());
  return result;
}

bool Own_Below_Threat() {
  return (Own_Tracked_Alt <= Other_Tracked_Alt);
}
bool Own_Above_Threat() {
  return (Other_Tracked_Alt <= Own_Tracked_Alt);
}
```

(a) Snippet of Tcas version 10

```
bool Own_Below_Threat() {
  /** DirectFix: replaced <= with <. **/
  return (Own_Tracked_Alt < Other_Tracked_Alt)
}

bool Own_Above_Threat() {
  /** DirectFix: replaced <= with <. **/
  return (Other_Tracked_Alt < Own_Tracked_Alt);
}
```

(b) A DirectFix repair (identical with the ground truth repair)

```
bool Non_Crossing_Biased_Climb() {
  int upward_preferred;
  int upward_crossing_situation;
  bool result;
  upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
  if (upward_preferred)
    result = !(Own_Below_Threat()) || ((Own_Below_Threat()) &&
/** SemFix: replaces !(Down_Separation >= ALIM())) with the following. **/
              (!(Other_RAC < Own_Tracked_Alt)));
  else
    result = Own_Above_Threat() &&
            (Cur_Vertical_Sep >= MINSEP) && (Up_Separation >= ALIM());
  return result;
}
```

(c) A SemFix repair

Fig. 9. Comparison of repairs for Tcas version 10

b) Qualitative Comparison with SemFix: Lastly, we provide a couple of concrete examples of repairs generated by DirectFix and SemFix. Fig. 9(b) shows a DirectFix repair from Tcas (buggy) version 10. Despite that two program locations are modified, the overall repair is simple; only two operators are replaced. This repair is identical with the ground

```
bool locate(character c, char *pat, int offset) {
    int i; bool flag = false;
    i = offset + pat[offset];
    while (i > offset)
        if (c == pat[i]) { flag = true; i = offset; }
        else i = i - 1;
    return flag;
}

bool omatch(char *lin, int *i, char *pat, int j) {
  ...
  if ((lin[*i] != NEWLINE) && (!locate(lin[*i], pat, j)))
  ...
}
```

(a) Snippet of Replace version 26

```
bool omatch(char *lin, int *i, char *pat, int j) {
  ...
  /** DirectFix: replace parameter j with j+1. **/
  if ((lin[*i] != NEWLINE) && (!locate(lin[*i], pat, j + 1)))
  ...
}
```

(b) A DirectFix repair (identical with the ground truth repair)

```
bool locate(character c, char *pat, int offset) {
    int i; bool flag = false;
    i = offset + pat[offset];
    while (i > offset)
        /** SemFix: replace c == pat[i] with i < 6. **/
        if (i < 6) { flag = true; i = offset; }
        else i = i - 1;
    return flag;
}
```

(c) A SemFix repair

Fig. 10. Comparison on repairs for Replace version 26

truth repair. Meanwhile, Fig. 10 shows two different repairs from DirectFix and SemFix for Replace (buggy) version 26. DirectFix successfully found the simple ground truth repair; it replaces a function parameter j with j+1 of function locate. Meanwhile, SemFix repaired function locate itself by changing an if-guard c==pat[i] to i<6. Although the repair is valid for the given test suite, this destructive repair causes a regression.

As compared with SemFix, DirectFix repairs are simpler, more frequently identical with the ground truth repairs, and less frequently cause a regression error.

VIII. Threats to Validity

Our subject programs mostly require small changes for repair. While software mining research shows that small fixes are abundant in the field [4], [23], some fixes inevitably require more sizable changes. In such situations, time would be exhausted before DirectFix can find a repair. Our subject programs do not represent such scenario.

In our experiments, we assumed that developers have insight about which function might be buggy. If an incorrect function is designated as buggy, DirectFix cannot generate the the ground truth repair. Our conjecture is that our repair method is better suited for fine-tuning a program and looking for a small fix, whereas search-based methods such as GenProg have advantage in their scalability. Combining these two contrasting methods seems possible; for example, after a search-based method aggressively narrows down the search space, DirectFix should be able to find the smallest patch in that reduced search space at a subsequent phase.

IX. Related work

A large volume of research has been conducted on automatic program repair. A number of them repair specific defect types [24]–[30], while DirectFix is designed to be a general purpose program repair tool. Unlike specifications-based methods [31]–[36], DirectFix falls into the category of the test-driven method whose goal is to find a patch that makes all tests in the given test suite pass. GenProg [5] and JAFF [37] use genetic programming (GP) to search for a patch. Using GP, statements can be deleted or moved. The fitness function of GP guides such mutations towards a patch. It was empirically shown that this approach scales to large programs such as PHP [12]. However, it often generates nonsensical patches, as pointed out in [7], due to its inherent nature of random mutation. To alleviate this problem, MUT-APR [38] mutates only pre-selected binary operators thereby restricting the defect types it can handle, and PAR [7] uses fix templates mined from actual human patches, instead of GP.

Meanwhile, SemFix [6] synthesizes a patch by semantic program analysis via dynamic symbolic execution, instead of performing syntactic search. Similar to DirectFix, SemFix also synthesizes a patch at the expression level by using component-based program synthesis [13]. More recently, Nopol [39] also took this semantic approach to fix control-related bugs (e.g., buggy if conditions). AutoFix [40] exploits contracts such as pre/post-conditions to generate random tests, localize faults, and generate a repair.

In all existing repair methods, fault localization is performed upfront before looking for a patch. We in contrast combine fault localization and repair generation, and as a result obtain the unique capability to take into account the simplicity of a patch. For this purpose, we exploit partial MaxSMT (pMaxSMT). Similarly, BugAssist [16] exploits pMaxSAT for fault localization. However, unlike in DirectFix, pMaxSAT is not used for repair synthesis – BugAssist does not fuse fault localization and repair generation.

X. Conclusion

In this paper, we have proposed DirectFix as a method that generates the simplest repair, following the thesis that small patches are easier to inspect and introduce less regressions (hence safer). DirectFix is a semantic analysis based repair method which differs from all existing repair methods in its integration of fault localization with repair generation. We have shown how those two phases can be integrated based on partial MaxSMT and component-based program synthesis. Patches produced by DirectFix are found to be simpler and safer than those produced by SemFix, the state-of-the-art semantic analysis based repair method.

ACKNOWLEDGEMENTS

This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

References

[1] M. E. Fagan, "Design and code inspections to reduce errors in program development." *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[2] J. Feller and B. Fitzgerald, *Understanding Open Source Software Development*. Addison-Wesley, 2001.

[3] P. Weißgerber, D. Neu, and S. Diehl, "Small patches get in!" in *MSR*, 2008, pp. 67–76.

[4] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *IEEE Trans. Software Eng.*, vol. 31, no. 6, pp. 511–526, 2005.

[5] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*, 2009, pp. 364–374.

[6] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: program repair via semantic analysis," in *ICSE*, 2013, pp. 772–781.

[7] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE*, 2013, pp. 802–811.

[8] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *ISSTA*, 2012, pp. 177–187.

[9] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization of test information to assist fault localization," in *ICSE*, 2002, pp. 467–477.

[10] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *PLDI*, 2003, pp. 141–154.

[11] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006)*, 2006, pp. 39–46.

[12] C. L. 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, pp. 3–13.

[13] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *ICSE*, 2010, pp. 215–224.

[14] L. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008, pp. 337–340.

[15] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *Theorem Proving in Higher Order Logics*, 2009, pp. 23–42.

[16] M. Jose and R. Majumdar, "Cause clue clauses: error localization using maximum satisfiability," in *PLDI*, 2011, pp. 437–446.

[17] E. Ermis, M. Schäf, and T. Wies, "Error invariants," in *FM*, 2012, pp. 187–201.

[18] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO*, 2005, pp. 364–387.

[19] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard version 2.0," SMT-LIB, Tech. Rep., 2012.

[20] Z. Fu and S. Malik, "On solving the partial max-sat problem," in *SAT*, 2006, pp. 252–265.

[21] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[22] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.

[23] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *ASE*, 2007, pp. 433–436.

[24] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *SOSP*, 2009, pp. 87–102.

[25] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *PLDI*, 2011, pp. 389–400.

[26] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard, "Detecting and escaping infinite loops with Jolt," in *ECOOP*, 2011, pp. 609–633.

[27] 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, pp. 233–244.

[28] A. Smirnov and T. cker Chiueh, "Dira: Automatic detection, identification and repair of control-hijacking attacks," in *Network and Distributed System Security Symposium*, 2005.

[29] S. Sidiroglou and A. D. Keromytis, "Countering network worms through automatic patch generation," *IEEE Security and Privacy*, vol. 3, no. 6, pp. 41–49, Nov. 2005.

[30] G. Novark, E. D. Berger, and B. G. Zorn, "Exterminator: Automatically correcting memory errors with high probability," *Commun. ACM*, vol. 51, no. 12, pp. 87–95, Dec. 2008.

[31] H. He and N. Gupta, "Automated debugging using path-based weakest preconditions," in *FASE*, 2004, pp. 267–280.

[32] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *CAV*, 2005, pp. 226–238.

[33] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using SAT," in *TACAS*, 2011, pp. 173–188.

[34] R. Könighofer and R. Bloem, "Automated error localization and correction for imperative programs," in *FMCAD*, 2011, pp. 91–100.

[35] H. Samimi, E. D. Aung, and T. Millstein, "Falling back on executable specifications," in *ECOOP*, 2010, pp. 552–576.

[36] B. Elkarablieh and S. Khurshid, "Juzi: a tool for repairing complex data structures," in *ICSE*, 2008, pp. 855–858.

[37] A. Arcuri, "Evolutionary repair of faulty software," *Appl. Soft Comput.*, vol. 11, no. 4, pp. 3494–3514, 2011.

[38] F. Y. Assiri and J. M. Bieman, "An assessment of the quality of automated program operator repair," in *ICST*, 2014, pp. 273–282.

[39] F. Demarco, J. Xuan, D. L. Berre, and M. Monperrus, "Automatic repair of buggy if conditions and missing preconditions with SMT," in *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, CSTVA 2014*, 2014, pp. 30–39.

[40] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Trans. Software Eng.*, vol. 40, no. 5, pp. 427–449, 2014.