SEMANTIC PROGRAM REPAIR

SERGEY MECHTAEV

NATIONAL UNIVERSITY OF SINGAPORE

2018

SEMANTIC PROGRAM REPAIR

SERGEY MECHTAEV

Specialist Diploma, Saint Petersburg State University

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE
2018

Supervisor:
Abhik Roychoudhury, Professor

Examiners:
Joxan Jaffar, Professor
Chin Wei Ngan, Associate Professor
Westley Weimer, Professor, University of Michigan

DECLARATION

Sergey Mechtaev

6 April 2018

ACKNOWLEDGEMENT

I thank my parents,
Vladimir Mechtaev and Olga Mechtaeva,
for giving me that, without which the rest would not have mattered.

I thank my friends,
Tan Shin Hwei and Xie Peichu,
for giving me more than I was able to accept.

I thank my colleagues and collaborators, (in alphabetical order)
Abhijeet Banerjee, Priyanka Bose, Marcel Böhme, Alessandro Cimatti, Dipanjan Das, Zhen Dong, Xiang Gao, Alberto Griggio, Lars Grunske, Manh-Dung Nguyen, Yannic Noller, Van-Thuan Pham, Shiqi Shen, Shaila G Sri, Edwin Lesmana Tjiong, Jooyong Yi, Xiao Liang Yu, Yulis,
for giving me what I would not have achieved alone.

I thank my supervisor,
Abhik Roychoudhury,
for giving me exactly what I came to him for, and even more.

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.

Contents

Summary

Debugging consumes significant amount of resources in software development projects. The inadequacy of used debugging techniques costs the global economy billions of dollars annually. Automated program repair is a promising technology that can reduce the cost of debugging by automatically eliminating program defects.

Early test-driven program repair techniques that scaled to large real-world programs utilized syntactic search without comprehending the meaning of the program and the defect. Although such techniques demonstrated encouraging results, they suffer from several limitations. First, since a test suite is an incomplete specification, automatically generated patches may not correspond to user intentions but merely overfit the tests. Secondly, syntactic techniques scale to relatively small search spaces and therefore can address only a small number of defects.

This work introduces a series of techniques to address the aforementioned challenges of automated program repair. These techniques are united by the idea of revealing the underlying program structure by means of semantic analysis. First, we propose an approach of encoding the repair problem as an instance of maximum satisfiability problem by reusing existing program synthesis and error diagnosis methods. Secondly, we devise a concise semantic signature that scales constraint-based repair to large real-world programs and that is capable of representing complex program changes. Third, we suggest an approach to increase the quality of generated patches by inferring missing specification from a reference implementation. Finally, we introduce symbolic execution with existential second-order constraints — an extension of symbolic execution that helps to alleviate the path explosion problem in the context of program repair.

Our experiments showed that the proposed techniques advance the state of the art of program repair. Semantic analysis helps to increase the quality of automatically generated patches. Apart from that, it enables program repair to scale to larger search spaces and consequently address more defect. We view these results as a step towards developing a general-purpose automated program repair system.

List of Tables

List of Figures

Chapter 1

Introduction

Program debugging is an important part of software development and maintenance. Debugging consumes significant percentage of resources for most software development projects. With the growth of the complexity of software systems, the number of bugs increases. The inadequacy of the used debugging techniques has lead to serious economic problems [102]. The dominant reason for the high cost of debugging is that this task is performed largely manually.

Defect (or bug) is defined as the cause of a software failure, where failure is a non-fulfillment of a given requirement. Program debugging is the task of localizing and correcting faults [97]. Program debugging involves a large number of activities that can vary depending on the used methodologies and tools, but the main activities that constitute debugging are fault localization (or isolation), fault understanding and fault correction. Fault localization tries to identify the parts of the program that are responsible for the failure. Fault understanding implies comprehending the cause of the failure. Fault correction is a modification of the program that eliminates the fault [77].

Due to the importance of the debugging problem, various debugging techniques has been investigated by researchers. While bug localization techniques have been extensively studied, they have not gained wide adoption in industry [77]. One of the reasons of their poor adoption is that they are typically focused on just providing a ranked list of faulty statements leaving the burden of understanding and repairing the faulty code to developers.

Automated program repair is an emerging area of research whose goal is to reduce the cost of debugging by automatically suggesting fixes. In the other words, it takes account of the entire debugging process: fault localization, fault understanding and fault correction. This can have significant impact on the way developers debug and repair software and potentially lead to wider adoption compared with automated debugging techniques. Besides, automated program repair opens new opportunities such as developing self-healing systems, which automatically detect incorrect behaviour and repair it by modifying its own code. This might find applications in such fiels as autonomous vehicles and the Internet of Things. Therefore, it is worthwhile to investigate automated program repair techniques.

1.1 Problem definition

Automated program repair techniques automatically and semi-automatically modify a given buggy program in order to eliminate a given defect. The presence of a defect is identified using given correctness criteria such as a test suite of a formal specification.

Fixing a defect may require implementing complex algorithms or making significant modifications to the source code, which makes the problem intractable in the general case. To make automated program repair practical, existing program repair approaches solve a simpler task. Specifically, they search for a patch that satisfies given correctness criteria in a given space of modifications.

More precisely, for a given program p with expressions $e_1, ..., e_n$ and a test suite T such that there is at least one test in T on which p fails, we consider the problem of finding a set of expressions $e'_1, ..., e'_n$ such that a program $p' := p[e_1 \mapsto e'_1, ..., e_n \mapsto e'_n]$, obtained by substituting the expressions $e_1, ..., e_n$ with the expressions $e'_1, ..., e'_n$, passes all the tests in T. There might be many modifications that pass the tests and, since a test suite is an incomplete specification, some of them might not correspond to the developer intentions. Thus, the definition can be augmented with additional criteria for choosing a patch such as a cost function that assigns lower cost to better patches.

The basic *generate-and-validate* repair algorithm [85] takes a buggy program and a test suite containing at least one failing test as inputs and produces a plausible patch as the output. In detail, it (1) generates a search space of candidate patches and (2) evaluates the candidates one-by-one by executing tests until it finds a patch that passes the whole test suite. Most of existing test-driven program repair techniques can be thought of as improvements upon this baseline algorithm. Particularly, they seek to (1) populate search space with effective patches, (2) assigning higher priority to quality repairs, and (3) increase the exploration speed.

1.2 Challenges

Exsiting program repair techniques have significant limitations. These limitations are caused by two main conceptual challenges:

Tractability Program repair techniques should scale to large real-world programs. Apart from that, program repair approaches search for repairs in huge spaces of candidate patches. Therefore, such system should provide mechanisms to address search space explosion.

Precision Since most of existing techniques rely on tests as the correctness criteria and tests is an incomplete specification, automatically generated patches may not satisfy user intentions, but overfit the tests [95]. Patches that merely pass given tests are referred to as *plausible* in literature [85]. Automated program repair system should either provide formal correctness guarantees or maximize the probability of finding correct repairs.

Other challenges of program repair include:

Quality Apart from correctness, read world software development project pose additional requirements such as code clarity and maintainability. Therefore, these criteria should be taken into consideration by automated program repair systems.

Speed Automated program repair systems should find patches within a given time budget.

Interactivity Since debugging is often an interactive process, automated program repair should provide interactive repair capabilities.

Expressiveness Since it is impossible to address arbitrary kinds of defects, automated program repair approaches should identify classes of defects that are important and tractable.

Applications In order for automated program repair to be economically viable, it is necessary to identify concrete usage scenarios.

Usability Automated program repair system should provide a convenient interface for developers.

1.3 Semantic repair

Early test-driven program repair techniques such as GenProg [56] that scaled to large real-world programs utilized search in a space of patches without comprehending the meaning of the program and the defect. We refer to such techniques as *syntactic*. Although such techniques demonstrated encouraging results, they suffer from limitations in precision and scalability. Since syntactic techniques do not comprehend the meaning of the defect and the program, they have only restricted means of ensuring correctness of generated patches, which leads to test overfitting. Since the space of syntactic changes is huge for large real-world programs, syntactic techniques are able to address only a small number of defects.

Contrary to syntactic techniques, semantics-based repair [73] aims to reveal the underling meaning of the program and the defect. First, this can

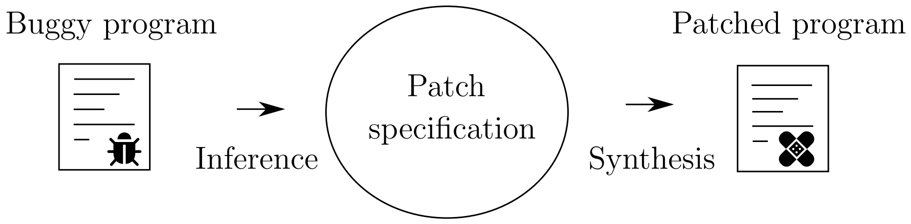


Figure 1.1: Semantic program repair workflow.

help to navigate the conceptually large search space. Secondly, semantics-based program repair provides excellent opportunities for improving patch correctness of generated patches. Typically, semantics-based techniques techniques split patch generation into specification inference and patch synthesis as shown in Figure 1.1. The inferred specification represent the meaning of the defect and therefore might be used to provide additional correctness guarantees.

Despite their benefits, previous semantic techniques suffered from important limitations. First, complex logical reasoning employed by such techniques limited their scalability to large real-world software. Secondly, semantic techniques based on path exploration [73] suffered from path explosion problem that restricted their effectiveness. Finally, since previous techniques inferred specification from tests, this specification only captured the property of "passing the tests". Thus, semantic techniques also suffer from test overfitting problem.

The goals of this thesis is to improve existing and develop new semantic analysis techniques for automated program repair to address its major challenges such as tractability and precision. Specifically, we propose a cohesive semantic program repair framework that consists of the following tightly integrated components:

- An approach of encoding the repair problem as a instance of satisfiability problem by reusing existing program synthesis and error diagnosis methods. We demonstrate that this method is able to generate complex multi-line repairs and provide higher quality repairs compared with previous techniques.

- An approach to scale semantic program repair to large real-world programs. To ensure its scalability, we devise a concise semantic signature whose size is independent on the size of the program and yet it is capable of representing complex multi-line program changes.

- An approach of inferring missing specification from a reference implementation in order to increase the quality of generated patches and provide additional correctness guarantees.

- Symbolic execution with existential second-order constraints — an extension of symbolic execution that helps to address the path explosion of symbolic execution in the context of program repair.

Our techniques impact the current state of practice by assisting developers in fixing defects. Specifically, the developed system enables developers to automatically and efficiently repair large number of defects in real-world software. It also provides additional guarantees that increase the probability that the automatically found patches are correct. This can impact developers productivity and increase the quality of software.

The thesis provides the following implications for future research: (1) semantic techniques based on constraint solving can scale to large real-world software and generate non-trivial patches and (2) semantic analysis techniques provide additional means of increasing correctness and quality of generated patches.

Chapter 2

Background

In this chapter, we recap the essential background knowledge for this thesis including satisfiability solving, symbolic execution, debugging, program synthesis and program repair.

2.1 Satisfiability

Propositional satisfiability (SAT) is the problem of determining whether a given propositional formula such as $(\alpha_1 \vee \neg\alpha_2) \wedge \alpha_3$ is satisfiable. Particularly, the problem consists in identifying if there exists an assignment of formula variables that makes the formula *True*, and also finding such a satisfying assignment. Although this problem is NP-complete [19], efficient algorithms such as CDCL [93] have been proposed that can handle large complex formulas in practice. Indeed, recent advances in SAT solving have enabled new powerful techniques for software development.

In many applications, it is inconvenient and inefficient to encode problems through propositional formulas. *Satisfiability modulo theories (SMT)* [9] addresses this by deciding the satisfiability of quantifier-free formulas such as $(\alpha_1 > 1) \wedge (\alpha_2 + \alpha_3 > 2)$ w.r.t. given background theories. Commonly employed background theories include the theories of integer linear arithmetic, bitvectors and arrays. Although state-of-the-art solvers such as Z3 [21] and MathSAT [17] realize sophisticated decision procedures to solve formulas over various background theories, there are theoretical considerations that limit their capabilities (e.g. non-linear integer arithmetic is undecidable [65]).

Each propositional formula can be transformed into an equisatisfiable formula in CNF form (a conjunction of clauses, where each clause is a disjunction of variables or their negations) using e.g. Tseytin algorithm [105]. *Maximum satisfiability (MAX-SAT)* is a generalization of SAT whose objective is to find the maximum number of clauses in a CNF formula that can be satisfied. *Maximum satisfiability modulo theories (MAX-SMT)* is a generalization of MAX-SAT for SMT. *Partial maximum satisfiability (Partial MAX-SAT/Partial MAX-SMT)* is a generalization of MAX-SAT/MAX-SMT in which some of the formula clauses are marked as hard and some are marked as soft. Then, the goal is to find the maximum number of soft clauses that are consistent with the hard clauses. One of the approaches to solve (Partial) MAX-SAT/MAX-SMT is to iteratively invoke a SAT/SMT solver to determine soft clauses that can be relaxed, as in Fu-Malik algorithm [31].

In this thesis, we consider formulas and terms built from predicate and function symbols (e.g. "$+$", "$-$", "$>$") from a given signature Γ. We denote the set of all such formulas and terms as L_Γ. We also consider a background theory $\mathcal{T}$ that fixes the interpretations of the symbols in Γ. We use the letters α, β, γ and δ to denote variables from L_Γ, and the letters π, ϕ and ψ

to designate formulas from L_Γ.

Assume that $\{\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k\}$ is an assignment of the variables from L_Γ (a mapping from the variables to values). We say that this assignment *satisfies* a formula π iff a substitution of the variables α_i with the corresponding values n_i (denoted as $[\![\pi]\!]_{\{\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k\}}$) evaluates to *True*.

2.2 Programs

We consider programs written in an imperative programming language. Programs are denoted as p_1, ..., p_k and the set of all program as $\mathcal{P}$. We define $p[e \mapsto e']$ as a program obtained from p by substituting an expression e with e'. Program variables are represented as $v_1, ..., v_k$ and the set of all program variables as $\mathcal{V}$. The considered programming language contains a statement `assume` defined as follows:

$$\texttt{assume}(\phi) \coloneqq \texttt{if } (\neg\phi) \; \{ \; \texttt{LOOP : goto LOOP;} \}$$

that is this statement triggers a non-termination (an infinite loop) when the given condition does not hold.

Concrete program states (functions from program variables to values) are indicated as σ and the set of all concrete program states is denoted as Σ; two concrete program states σ_1 and σ_2 are equal iff $\forall v \in \mathcal{V}.\ \sigma_1(v) = \sigma_2(v)$. We indicate a program state obtained from σ by updating the value of the variable v to n as $\sigma[v \mapsto n]$. The value of an expression e evaluated in the context σ is denoted as $[\![e]\!]_\sigma$. We define a concrete program execution as in the following:

Definition 1 (Concrete execution). *A concrete execution procedure* $Exec : \mathcal{P} \times \Sigma \rightarrow \Sigma \cup \{\omega\}$ *is a function that for a given program* p *and a concrete input state* σ_{in} *returns the corresponding output state* σ_{out} *if the program terminates, and the literal* ω *otherwise.*

Definition 2 (Partial equivalence under σ). *Let* p_1 *and* p_2 *be programs,* σ *be a program state. We say that* p_1 *and* p_2 *are* partially equivalent *under* σ *iff at least one of the following holds:*

- $Exec(p_1, \sigma) = \omega$;
- $Exec(p_2, \sigma) = \omega$;
- $Exec(p_1, \sigma) = Exec(p_2, \sigma)$.

Conditional equivalence [46] is a relaxed notion of partial equivalence that checks equivalence only for a subset of inputs.

Definition 3 (Conditional partial equivalence). *Let* p_1 *and* p_2 *be programs,* $\phi \in L_\Gamma$ *be an input condition. We say that* p_1 *and* p_2 *are* conditionally partially equivalent *under an input condition* ϕ *iff they are partially equivalent under each input in* $\{\sigma \mid [\![\phi]\!]_\sigma = True\}$.

2.3 Symbolic execution

Symbolic execution is a powerful program analysis technique that relies on SMT solving. Having been originally proposed for software testing [49, 13], it has since found a wide range of other applications in software engineering including software verification [37], program debugging [82] and program repair [73]. In symbolic execution, program inputs are assigned symbolic variables instead of concrete values. The result of executing a program with symbolic inputs is a set of constraints over these symbolic variables called path conditions. The path condition of a program path captures all inputs that would drive the execution along this program path. SMT solvers are used in symbolic execution engines to generate concrete program inputs by solving path conditions and to determine the feasibility of paths (if there is at least one input that drives the execution along the considered path).

Symbolic execution operates *symbolic program states.* We use the letter θ to indicate symbolic program states, that is functions from program variables to logical terms from L_Γ (for a program variable v, the corresponding logical term is $\theta(v)$), the set of all symbolic program states is denoted as Θ. We express the equality of two symbolic program states θ_1 and θ_2 as the formula $\theta_1 = \theta_2 \coloneqq \bigwedge_{v \in \mathcal{V}} \theta_1(v) = \theta_2(v)$. We indicate a symbolic program state obtained from θ by updating the value of the variable v to ϕ as $\theta[v \mapsto \phi]$. We denote a logical term computed by evaluating a program expression e in the context θ as $[\![e]\!]_\theta$. We also introduce a concretization of symbolic states defined as follows:

Definition 4 (Concretization)**.** *Let θ be a symbolic program state, $\{\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k\}$ be an assignment of the variables from L_Γ. A concrete state $[\![\theta]\!]_{\{\alpha_1 \mapsto n_1,...,\alpha_k \mapsto n_k\}}$ is the* concretization *of θ with the assignment $\{\alpha_1 \mapsto$*

ALGORITHM 1: Symbolic interpreter

```
Procedure symInter(instruction pointer IP, symbolic state $\theta$, path condition $\pi$)
    I := getInstruction($IP$);
    switch $I$ do
        case Assignment v := e do
            $\theta' := \theta[v \mapsto [\![e]\!]_\theta]$;
            $IP' :=$ increment($IP$);
            symInter($IP'$, $\theta'$, $\pi$);
        end
        case Conditional if e then C_1 else C_2 do
            $\phi := \pi \wedge [\![e]\!]_\theta$;
            if isSatisfiable($\phi$) then
                $IP' :=$ getPointer($C_1$);
                symInter($IP'$, $\theta$, $\phi$);
            end
            ... // check the other branch
        end
        otherwise do
            ... // handle other instructions
        end
    end
```

$n_1, ..., \alpha_k \mapsto n_k\}$ is defined as follows:

$$[\![\theta]\!]_{\{\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k\}} := \lambda v.\ [\![\theta(v)]\!]_{\{\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k\}}$$

that is a mapping of program variables into values expressed using lambda notation, computed by substituting all the variables α_i in the logical terms in the codomain of θ with the corresponding values n_i.

In this thesis, we consider symbolic execution from two angles: an operational perspective that characterizes symbolic execution as a program analysis mechanism and a denotational perspective that characterizes symbolic execution as a program comprehension mechanism.

From the operational perspective, symbolic execution is a generalized program interpreter that maintains symbolic memory and path conditions as shown in Algorithm 1. The function `symInter` takes an instruction pointer, a symbolic program state and a path condition, performs symbolic execution of the corresponding instruction, and recursively continues execution. For an assignment `v` $:=$ `e`, this function updates the state by replacing the value of `v` with `e` evaluated in the context θ (denoted as $[\![e]\!]_\theta$). Then, the execution continues from the next instruction. For a conditional `if e then` $\mathtt{C}_1$ `else` $\mathtt{C}_2$, this function checks whether the if-condition is consistent with the current path condition and whether the negation of the if-condition is consistent with the path condition (the later case is omitted). If the constraint is satisfiable, the algorithm continues execution of the corresponding branch with an augmented path condition.

From the denotational perspective, symbolic execution is an approach to extract specification or summary of program behaviour. For this, we define symbolic execution non-constructively as follows:

Definition 5 (Symbolic execution)**.** *Symbolic execution SymExec* : $\mathcal{P} \times \Theta \to 2^{L_\Gamma \times \Theta}$ *is a function that for a given program p and a symbolic input state* θ_{in} *returns a finite set of pairs* $\{(\pi, \theta_{out})\}$*, where* π *is the path condition and* θ_{out} *is the corresponding symbolic output state. For each assignment of the symbolic variables* $\{\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k\}$*, if this assignment satisfies the formula* π*, then* $\sigma_{out} = \mathit{Exec}(p, \sigma_{in})$*, given that* $\sigma_{in} := [\![\theta_{in}]\!]_{\{\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k\}}$ *and* $\sigma_{out} := [\![\theta_{out}]\!]_{\{\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k\}}$.

Obviously, *SymExec* can be implemented using the procedure `symInter` presented in Algorithm 1.

Consider the function `search` in Figure 2.1a. This function takes an array

```
size_t search(int data[],
              size_t len,
              int (*pred)(int)) {
  size_t i;
  for (i = 0; i < len; i++)
    if (pred(data[i]))
      return i;
  return len;
}
```

(a) Search function.

Executing `search` with symbolic inputs $\alpha_1, \alpha_2, \alpha_3$:

```
int pos(int x) { return x > 0; }
search((int[]){α1,α2,α3}, 3, pos);
```

Symbolic input state:

$$\theta_{in} := \{\texttt{data}[0] \mapsto \alpha_1, \texttt{data}[1] \mapsto \alpha_2, \texttt{data}[2] \mapsto \alpha_3, \texttt{len} \mapsto 3\}$$

Symbolic execution results:

Path condition π	Generated input	Output state θ_{out}
$\alpha_1{>}0$	$\{\alpha_1 \mapsto 1, \alpha_2 \mapsto 0, \alpha_3 \mapsto 0\}$	$\{\texttt{return} \mapsto 0, ...\}$
$\alpha_1{\le}0 \wedge \alpha_2{>}0$	$\{\alpha_1 \mapsto 0, \alpha_2 \mapsto 1, \alpha_3 \mapsto 0\}$	$\{\texttt{return} \mapsto 1, ...\}$
$\alpha_1{\le}0 \wedge \alpha_2{\le}0 \wedge \alpha_3{>}0$	$\{\alpha_1 \mapsto 0, \alpha_2 \mapsto 0, \alpha_3 \mapsto 1\}$	$\{\texttt{return} \mapsto 2, ...\}$
$\alpha_1{\le}0 \wedge \alpha_2{\le}0 \wedge \alpha_3{\le}0$	$\{\alpha_1 \mapsto 0, \alpha_2 \mapsto 0, \alpha_3 \mapsto 0\}$	$\{\texttt{return} \mapsto 3, ...\}$

(b) Symbolic execution.

Figure 2.1: Testing search function via symbolic execution.

`data`, a value `len` representing its length, a pointer to a predicate function `pred`, and returns the index of the first element of the array that satisfies the predicate.

In symbolic execution, numeric inputs are replaced with logical variables as shown for the elements $\alpha_1, \alpha_2, \alpha_3$ of the array in Figure 2.1b. Assume that the predicate `pred` is a function `pos` that checks if a given value is positive.

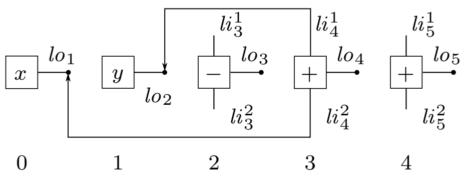
In this context, symbolic execution explores four paths as shown in the table in Figure 2.1b, in which the path conditions π are constraints over the variables $\alpha_1, \alpha_2, \alpha_3$, and the output states contain a special variable `return` that captures the return value of the function. The test inputs generated for each path by solving the corresponding path condition are shown in the column "Generated input".

2.4 Program synthesis

Program synthesis is a methodology for automatically constructing programs that satisfy given requirements. From the logical point of view, program synthesis can be considered as a second-order constraint solving [20]. For example, for a given set of tests $t_1, ..., t_n$, where each test $t_i := (\sigma_i, r_i)$ is a pair of input state σ_i and an output value r_i, program synthesis problem can be expressed as a second-order satisfiability problem over terms e:

$$\exists e \in L_\Gamma.\ \bigwedge_i [\![e]\!]_{\sigma_i} = r_i \tag{2.1}$$

The most trivial approach to solve the above formula is to enumerate all possible terms until we find a term that satisfies the formula. However, this approach might have performance limitations due to the large search space. In this thesis, we focus on synthesis algorithms that reduce this problem to SMT solving.



(a) Circuit of term $x + y$ (below are locations).

$lo_1, lo_2 = 0, 1$
$lo_3, lo_4 = 2, 3$
$li_4^2, li_4^1 = 0, 1$

(b) L-variables.

Figure 2.2: Encoding space of terms via integer location variables.

2.4.1 SMT-based component-based program synthesis

Jha et al. [38] proposed to solve the formula (2.1) by semantically encoding a space of terms using linear integer arithmetic constraints. In this approach, terms are represented as circuits built from user-provided components such as addition, subtraction, etc. Connections between components are captured using integer *location variables*. For each component, location variables are introduced for its output and all its inputs. If the location of an input of one component coincides with the location of the output of another component, they are considered to be connected. Thus, given a valuation of location variables, we can reconstruct a program from the connected components. The goal of an SMT solver in this case is to find a valuation of location variables that corresponds to a program that passes all the tests.

Assume that $\texttt{out}_i$ is the output of i-th component, lo_i is the location of the output of the i-th component, $\texttt{in}_i^j$ is the j-th input if the i-th component, li_i^j is the locations of the j-th input if the i-th component, C is the number of components, N_i is the number of inputs of the i-th component, F_j is the semantics if the j-th component (e.g. $\lambda xy.\ x + y$ for addition). The set of well-formed terms is encoded using $\phi_{wpf} \coloneqq \phi_{range} \wedge \phi_{cons} \wedge \phi_{acyc}$, such that

$$\begin{aligned}
\phi_{range} &\coloneqq \bigwedge_{i\in[1,C]} \Big(0 \le lo_i < C \quad \wedge \bigwedge_{j\in[1,N_i]} 0 \le li_i^j < C\Big) \\
\phi_{cons} &\coloneqq \bigwedge_{i,j\in[1,C],i\neq j} \quad lo_i \neq lo_j \\
\phi_{acyc} &\coloneqq \bigwedge_{i\in[1,C],j\in[1,N_i]} \quad lo_i > li_i^j
\end{aligned}$$

where range constraints ϕ_{range} allocate inputs and outputs within a legal range, consistency constraints ϕ_{cons} ensure that all outputs have unique locations, and acyclicity constraints ϕ_{acyc} forbid loops. Besides, connection constraints ϕ_{conn} bind location variables and connections between components, and semantic constraints ϕ_{sem} define the relation between components' inputs and outputs:

$$\begin{aligned}
\phi_{conn} &\coloneqq \bigwedge_{i,j\in[1,C],k\in[1..N_i]} \quad lo_i = li_j^k \Rightarrow \mathtt{out}_i = \mathtt{in}_j^k \\
\phi_{sem} &\coloneqq \bigwedge_{i\in[1,C]} \mathtt{out}_i = F_i(\mathtt{in}_i^1, \mathtt{in}_i^2, ..., \mathtt{in}_i^{N_i})
\end{aligned}$$

A term is constructed from an assignment of locations variables that satisfies $\phi_{wpf} \wedge \phi_{conn} \wedge \phi_{sem}$ using a function `Lval2Term`. This function connects inputs and outputs of components that have the same location. For example, $\lambda xy.\ x+y$ is constructed from the assignment in Figure 2.2b (as in the circuit in Figure 2.2a).

2.5 Debugging

Since debugging consumes substantial amount of resources in practice, approaches to automated this process have been extensively studied. A group of debugging techniques is focused on fault localization, that is identifying the part of the program that is responsible for the failure.

Statistical fault localization [60, 2, 41] is a family of techniques based on statistical information, First, statistical fault localization methods count the number of times each statement is executed by passing and failing test cases. Then, they use this data to compute suspiciousness score for each statement. For example, Tarantula [42] relies on the following formula to compute the suspiciousness score:

$$score(stmt) := \frac{a_{ef}/(a_{ef} + e_{nf})}{a_{ef}/(a_{ef} + e_{nf}) + a_{ep}/(a_{ep} + e_{np})} \tag{2.2}$$

where a_{ef} is the number of failing tests that executed *stmt*, a_{nf} is the number of failing tests that did not execute *stmt*, a_{ep} is the number of passing tests that executed *stmt*, a_{np} is the number of passing tests that did not execute *stmt*. Finally, they rank statements according to their suspiciousness score. The benefit of these techniques is their simplicity and good scalability, since they require only computing statement coverage to calculate suspiciousness score. The drawback of these approaches is their imprecision, since they do not guarantee that the bug can be fixed at the identified locations.

BugAssist [43] is a fault localization tool that uses maximum satisfiability to identify suspicious statements. Specifically, it represents the program as a formula. This formulas is conjoined with the assignment of input variables to the test inputs and with the test assertions. After that, the formula is solved using a Partial MAX-SMT solver. Specifically, the test inputs and assertions are treated as hard constraints and program statements as soft constraints. The Partial MAX-SMT solver returns a subset of statements that are responsible for the bug. The benefit of this approach consists in the ability to identify multi-location bugs and in that is guarantees that the bug can be fixed by altering the identified locations. The drawbacks of this approach include its limited scalability, since it requires translating program into a logical formula, and the fact that the Partial MAX-SMT problem might have many solutions in practice, therefore the user might have to iterate through and manually inspect all these solutions.

2.6 Program repair

Automated program repair is an extremely challenging problem, since repairing a program might require implementing an arbitrary algorithm. Apart from that, automatic program repair requires providing a correctness criteria that might include both functional and non-functional requirements. As a result, existing repair methods are not able to fix arbitrary faults. Instead, they focus on a simpler task of searching for a patch in a pre-defined space of modifications that satisfy given correctness criteria such as a test suite and possible additional quality metrics.

2.6.1 Genetic programming based repair

Genetic algorithm is a search strategy that resembles the process of biological evolution. It maintains a population of individuals, and uses mutation and crossover operators and a fitness function. At each iteration, all individuals from current population are evaluated using the fitness function. The resulting value identifies the probability that they will reproduce the next population. Mutation and crossover operation are applied to each candidate to generate the next population. *Genetic programming* is a special case of genetic algorithm where the population consists of computer programs.

GenProg [56] is a program repair approach based on genetic programming. The fitness function utilized in GenProg counts the number of passing tests for a given program. New populations of programs are constructed by applying mutation operators are such as swap, delete and insert to the current population (the insert operator copies code fragments from the same program) as shown in Figure 2.3. Apart from that, GenProg uses statistical-based localization approach for mutating statements with higher suspiciousness score.

GenProg have several advantages and disadvantages. The main advantage of this approach is its scalability, since it has been applied to repair large complex programs. Apart from that, it is able to generate non-trivial multi-line fixes. The main disadvantages of GenProg are the limited number of bugs that can be repaired and the low quality of fixes. The first drawback is caused by the limitation of the search space: correct functionality must be located within the program to repair. The low quality of fixes is caused by

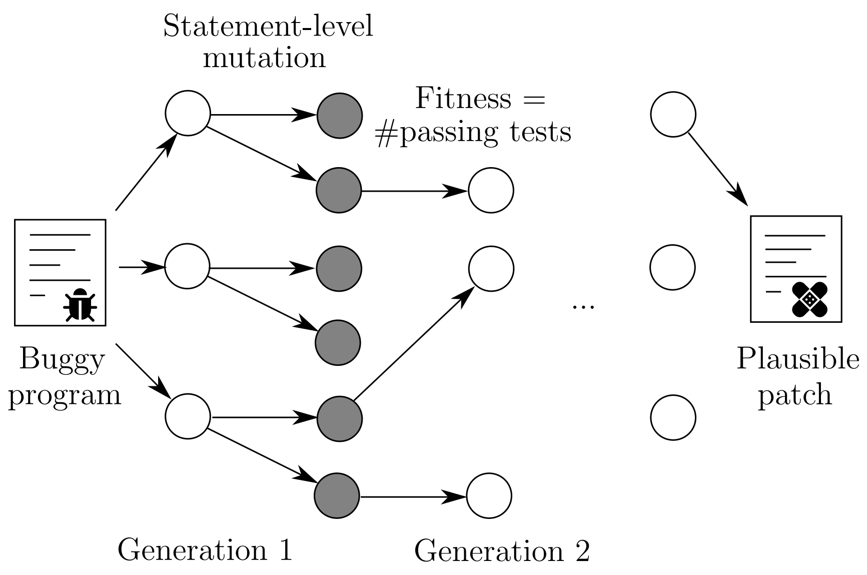


Figure 2.3: GenProg workflow.

the fact that this algorithm can make a lot of random modifications that can introduce new bugs and break the structure of the program.

2.6.2 Semantic analysis based repair

SemFix [73] is an approach to program repair based on semantic analysis. Its workflow consists of three parts: (1) fault localization, (2) specification inference using symbolic execution and (3) patch synthesis.

Consider an example of the buggy TCAS program in Figure 2.4a from SIR benchmark [24]. The intended behavior of this procedure corresponds to the following function:

```
is_upward_preferred(...) = inhibit*100+up_sep > down_sep
```

As can be seen, there is a fault at the line 5. The expression assigned to the variable `bias` is `down_sep` instead of `(up_sep + 100)`. Consider a test suite presented in Figure 2.4b that contains two failing test cases.

```
 1 int is_upward_preferred(int inhibit,
 2                         int up_sep,
 3                         int down_sep) {
 4   int bias;
 5   if (inhibit)
 6     bias = down_sep; //fix: bias=up_sep+100
 7   else
 8     bias = up_sep;
 9   if (bias > down_sep)
10     return 1;
11   else
12     return 0;
13 }
```

(a) Buggy function from TCAS.

`inhibit`	`up_sep`	`down_sep`	Expected output	Observed output
1	0	100	0	0
1	11	110	1	0
0	100	50	1	1
1	-20	60	1	0
0	0	10	0	0

(b) Passing and failing tests.

Line	Score	Rank
6	0.75	1
12	0.6	2
5	0.5	3
9	0.5	4
11	0	5
10	0	6

(c) Suspiciousness.

`inhibit`	`up_sep`	`down_sep`	π	Output
1	0	100	$\alpha > 100$	1
1	0	100	$\alpha \leq 100$	0
1	11	110	$\alpha > 110$	1
1	11	110	$\alpha \leq 110$	0
1	-20	60	$\alpha > 60$	1
1	-20	60	$\alpha \leq 60$	0

(d) Extracted specification.

Figure 2.4: Repairing function using SemFix.

To repair this program, SemFix first creates a ranking of suspicious statements using an existing statistical fault localization method. Specifically, it computes statement coverage for each tests, and applies the formula (2.2) to create a ranking shown in Figure 2.4c. Finally, it iterates through the ranked locations starting from the most suspicious statement.

For each suspicuous location, SemFix replaces the expression (the right-hand side of a assignment or a condition) with a symbolic variables. For example, for the statement in line 6, it replaces the expression `down_sep` with a fresh symbolic variable α. Then, for each test $\{\sigma_{in}, \sigma_{out}\}$ (a pair of input and output states) that covers the suspicuous location, it executes the program symbolically in such a way that the concrete test inputs σ_{in} are used as the symbolic input state. Thus, it executes the program concretely before the suspicuous location, and symbolically starting from the suspicuous location. As a result, it extracts specification for the suspicuous statement shown in Figure 2.4d. In this table, for each test that covers the suspicuous location, π represents a path condition, and the column "Output" represents the return value of the function computed along the corresponding path.

After inferring specification, SemFix uses this specification to synthesize a replacement for the considered expressions that would enable the program to pass the test. For tests $\{\sigma_{in}, \sigma_{out}\}_i$, and for the inferred pairs of a path condition and an output symbolic state $\{\pi^i, \theta^i_{out}\}_j$ for each test i, it solves the following second-order formula:

$$\exists e \in L_\Gamma.\ \bigwedge_i \bigvee_j \pi^i_j[\alpha \mapsto e] \wedge \sigma^i_{out} = \theta^i_{j\ out}$$

This formula states that for each test i, there should be at least one path π^i_j, along which the test passes if the symbolic variables α is replaced with the expression e. For the example in Figure 2.4d, SemFix solves a formula representing the three paths in Figure 2.4d that enable the program to produce the expected outputs (along which $\sigma^i_{out} = \theta^i_{j\ out}$):

$$\begin{aligned}\exists e \in L_\Gamma.\ & [\![e]\!]_{\{\texttt{inhibit}\mapsto 1,\ \texttt{up\_sep}\mapsto 0,\ \texttt{down\_sep}\mapsto 100\}} \leq 100 \\ & \wedge [\![e]\!]_{\{\texttt{inhibit}\mapsto 1,\ \texttt{up\_sep}\mapsto 11,\ \texttt{down\_sep}\mapsto 110\}} > 110 \\ & \wedge [\![e]\!]_{\{\texttt{inhibit}\mapsto 1,\ \texttt{up\_sep}\mapsto -20,\ \texttt{down\_sep}\mapsto 60\}} > 60\end{aligned}$$

Solving this formula can be seen as synthesizing a code fragment (unknown function) with properties of the function being gleaned from test executions. The formula is solved using component-based synthesis described in Section 2.4. For example, it might generate the expression `(up_sep + 100)` that satisfies this formula.

The advantage of SemFix is that it might potentially repair more defects, since it does not require that the correct functionality already exists in the source code, instead it synthesizes new code using program synthesis. However, SemFix has some limitations. First, SemFix extracts specification from tests, therefore it is subjected to the test overfitting problem so as other test-driven program repair approaches. This limitation is partially addressed by DirectFix [69] that synthesize minimal code transformations that are less likely to break unspecified functionality (Chapter 3), and by SemGraft [67] that extracts the missing specification from a reference implementation (Chapter 5). Secondly, SemFix inherits the limitations of symbolic execution such as the path explosion problem that might limit its effectiveness. This limitation is addressed by symbolic execution with existential second-order constraints [66] that helps to alleviate path explosion by taking the search space into account during path exploration (Chapter 6). Finally, SemFix extracts specification that effectively captures the semantics of the whole program, which limits its scalability when applied to large programs. This limitation is addressed by Angelix [70] that extracts concise synthesis specification, whose size is independent of the size of the program, and yet it is capable of capturing complex multi-line changes (Chapter 4).

Chapter 3

Program repair via maximum satisfiability

When repairing a program, it is preferable to construct patches which are simple and readable. This is because software maintainers would not blindly accept a suggested patch, but rather they would review and inspect a patch before accepting it [28, 29], for instance, to ensure that the patch resolves the problem and does not introduce new bugs. 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 [109, 81]. Hence, it is instructive to have program repair tools produce small patches. However, previous automatic repair tools such as GenProg [108] and SemFix [73] described in Section 2.6 do not explicitly take into account of the simplicity of a patch, although more general issues about patch quality (e.g., patch maintainability [30] and users' willingness to accept patches [48]) have been raised and studied.

Finding a simple repair 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 patch. Note that even for finding one repair, existing repair tools often take substantial amount of time. We propose in this chapter an efficient test-driven repair method (and its implementation DirectFix) that can find simple repairs. 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 (see Section 2.5) to choose program locations to modify. 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 fuse the fault localization and repair steps into a single step via partial MAX-SAT solving. The *main technical contribution* of this chapter 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 Partial MAX-SMT problem (see Section 2.1). For a given buggy program and a test suite, we construct a logical formula in a way that a satisfying assignment of this formula corresponds to the simplest repair — simplest in the sense that the structure of the original buggy program is maximally preserved. While the nature of MAX-SMT 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 [38].

We implement our approach into a tool, DirectFix, that formulates a necessary formula and solves it using our Partial MAX-SMT solver implemented on top of Z3 SMT solver [21]. 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 often make. Despite the limited size of our subject programs and the limitations inherited from the underlying tools upon which DirectFix is built — most notably, VCC [18], 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 [73] 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 less frequently than SemFix repairs (for which we checked against the test universe and not just the test suite used as the correctness criteria for the repair tools).

3.1 Motivating examples

We present three simple motivating examples in this section (in Section 3.5, we also present our repairs for actual programs). Consider the program snippet in Figure 3.1a. 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, Figure 3.1b and 3.1c 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 Figure 3.1b resembles a repair generated by GenProg [108]. GenProg grafts existing code onto a buggy program in an attempt of repair. As a result, GenProg often generates repairs that look complex to human developers, as pointed out in [48].

Meanwhile, a more recent repair tool, SemFix [73], 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 [73] performs repair at

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

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

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

(b) A repair that resembles a GenProg repair.

```
x = E1;
y = E2;
S1;
if (x >= y) // SIMPLE FIX:  >= is substituted for  >
  return 0;
else
  return 1;
```

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

Figure 3.1: DirectFix motivating example 1

the expression level, unlike GenProg that performs repair at the statement level or binary level [89]). However, SemFix still often generates repairs that are more complex than necessary. Figure 3.2 shows such an example. Given a buggy program in Figure 3.2a – the first two lines are mistakenly swapped, and the equal signs (=) are omitted –, SemFix can generate a repair shown in Figure 3.2b. Compare this repair with an alternative repair shown in Figure 3.2c. 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 Figure 3.3a that checks whether the character c is included in the string (character array) s. The table in Figure 3.3b 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 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. Figure 3.3c and 3.3d show two possible repairs – both repairs pass all the tests in Figure 3.3b. However, the first repair (Figure 3.3c) 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,

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

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

```
if (x > y)
  if (x > z)
     out = 10;
  else
     out = 20;
else out = 30;
return ((x>=z)? ((x>=y)? 10 : 20) : 30); // This line is one
possible repair.
```

(b) A repair that resembles a SemFix repair.

```
if (x >= z)    // SIMPLE FIX:  >=z  is substituted for  >y
  if (x >= y) // SIMPLE FIX:  >=y  is substituted for  >z
     out = 10;
  else
     out = 20;
else out = 30;
return out;
```

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

Figure 3.2: DirecFix motivating example 2

```
// 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.

Figure 3.3: DirectFix motivating example 3

what is a right test suite is another important research question that has not been thoroughly addressed yet.

Meanwhile, the second simpler repair (Figure 3.3d) 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 1. *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.

3.2 Overview

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 [38] extended to produce syntactically minimal changes as well as to improve scalability.

DirectFix utilizes program semantics expressed through a logical formula called *trace formula* in the literature [43, 27].

Definition 6 (Trace formula). *Let p be a program, $x_1, ..., x_n$ are variables of* *p. A formula* ϕ_p *over the variables* $\alpha_1, ..., \alpha_n, \beta_1, ..., \beta_n$ *is a trace formula of* p *if the following property holds:*

$$[\![\phi_p]\!]_{\{\alpha_1 \mapsto i_1, ..., \alpha_n \mapsto i_1, \beta_1 \mapsto o_1, ..., \beta_n \mapsto o_1\}} = \mathit{True} \Rightarrow \mathit{Exec}(p, \sigma_{\mathit{in}}) = \sigma_{\mathit{out}}$$

where $\sigma_{\mathit{in}} \coloneqq \{x_1 \mapsto i_1, ..., x_n \mapsto i_n\}$ *and* $\sigma_{\mathit{out}} \coloneqq \{x_1 \mapsto o_1, ..., x_n \mapsto o_n\}$ *.*

A trace formula can be constructed using the predicate transformer [23]. For example, Figure 3.4b demonstrates the trace formula for the function *foo* shown in Figure 3.4a. 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:

$$\mathcal{O} \coloneqq (\alpha_1 = 0) \wedge (\alpha_2 = 0) \wedge (\beta_3 = 3)$$

The conjunction $\varphi_{\mathit{buggy}} \wedge \mathcal{O}$ is unsatisfiable, reflecting 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 $\varphi_{\mathit{repair}}$ makes $\varphi_{\mathit{repair}} \wedge \mathcal{O}$ satisfiable. In our example, the ground truth repair is as follows:

$$\begin{aligned}\varphi_{\mathit{repair}} \coloneqq &(\text{if } (\alpha_1 \geq \alpha_2) \text{ then } (\beta_2 = \alpha_2 + 1) \text{ else } (\beta_2 = \alpha_2 - 1)) \\ &\wedge (\beta_3 = \beta_2 + 2)\end{aligned}$$

[1]If there are multiple tests, say two, we formulate $\mathit{Rename}(\varphi_{\mathit{buggy}} \wedge \mathcal{O}_1) \wedge \mathit{Rename}(\varphi_{\mathit{buggy}} \wedge \mathcal{O}_2)$, 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) Buggy function and failing test.

$$\begin{aligned}\varphi_{buggy} :=& (\text{if } (\alpha_1 > \alpha_2) \text{ then } (\beta_2 = \alpha_2 + 1) \text{ else } (\beta_2 = \alpha_2 - 1)) \\ & \wedge (\beta_3 = \beta_2 + 2)\end{aligned}$$

(b) The trace formula φ_{buggy} for `foo`; variables α_1 and α_2 correspond to the input values of the variables x and y, β_1 and β_2 correspond to the output values of the variables x and y, and β_3 to the return value of the program.

Figure 3.4: Trace formula of buggy program

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 Partial MAX-SMT problem.

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

$$\begin{aligned}\varphi_{rc} :=& (\text{if } v_1 \text{ then } (\beta_2 = v_2) \text{ else } (\beta_2 = v_3)) \wedge (\beta_3 = v_4)\\ &\wedge v_1 = \mathit{cmpnt}(\alpha_1 > \alpha_2) \wedge v_2 = \mathit{cmpnt}(\alpha_2 + 1)\\ &\wedge v_3 = \mathit{cmpnt}(\alpha_2 - 1) \wedge v_4 = \mathit{cmpnt}(\beta_2 + 2)\end{aligned}$$

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., $v_1 = \mathit{cmpnt}(\alpha_1 > \alpha_2)$) with applied function *cmpnt*. This function componentizes its parameter expression into a circuit form, following the idea of component-based synthesis described in Section 2.4.

To obtain the simplest repair, we use a Partial MAX-SMT solver. In Partial MAX-SMT, 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

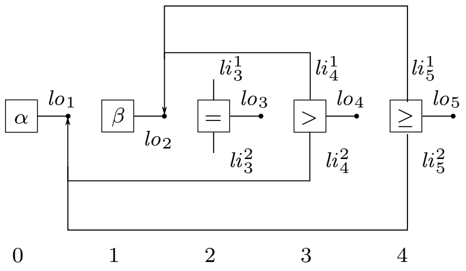


Figure 3.5: Repairing expression $\alpha > \beta$ by replacing $>$ with $\geq$.

the structure of the program expressions. Assume that the component α has the index 1, β has the index 2, $=$ has the index 3, $>$ has the index 4, and $\geq$ has the index 5. We construct the *structure constraint* for the expression $\alpha > \beta$ as follows:

$$li_4^1 = lo_2 \wedge li_4^2 = lo_1 \wedge lo = lo_4$$

where the variable lo is a designated variable that specifies the location of the output of the whole expression. 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 Partial MAX-SMT solver. Then, the solver removes some structure constraints (if necessary), and returns a model corresponding to a fix.

Figure 3.5 shows how a solver can modify the expression $\alpha > \beta$ using an additional component $\geq$ in order to repair the program. Specifically, it removes one connection between the outputs of $>$ and the output of the whole expression corresponding to the structure constraint $lo = lo_4$, 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 output of the whole expression (the variable lo). 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.

3.3 Methodology

Our approach combines fault localization and correction into a single step, which is achieved by reducing repair problem to Partial MAX-SAT problem. Unlike in component-based synthesis (Section 2.4.1), 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 formulate the following repair problem.

Definition 7 (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 2 describes how we generate a repair condition, given a test suite TS and a trace formula ϕ_p as input. We assume that each test $t \in TS$ is a part of input and output states $(\sigma_{in}, \sigma_{out})$, that the corresponding oracle constraint is

$$\mathcal{O}_t \coloneqq (\bigwedge_i \alpha_i = \sigma_{in}(x_i)) \wedge (\bigwedge_i \beta_i = \sigma_{out}(x_i))$$

where x_i are program variables.

Our algorithm substitutes fresh variables v_i for the rvalue expressions (the expressions of ϕ_p that are originated from the conditionals or right-hand-side expressions of a given buggy program[2]) to construct the formula $\phi_p[e_i \mapsto v_i]$. The formula $\phi_p[e_i \mapsto v_i] \wedge (\bigwedge_i v_i = e_i)$ is semantically equivalent to the initial trace formula ϕ_p. However, expressions e_i that we allow to modify are now distinguished from the rest of the formula. Algorithm 2 applies *cmpnt* to all the components of e_i together with additional component, and the formula $(\bigwedge_i v_i = \mathit{cmpnt}(e_i)) \wedge \phi_p[e_i \mapsto v_i] \wedge \mathcal{O}_t$ is returned as hard clauses of the repair condition for each test case $t \in TS$.

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. In the previous section, we showed that expression $\alpha > \beta$ is encoded into the following structure constraint ϕ_{struct} (assume that the component α has the

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

index 0, β has the index 1, $=$ has the index 2, $>$ has the index 3, and $\geq$ has the index 4):

$$li_3^1 = lo_1 \wedge li_3^2 = lo_0 \wedge lo = lo_3$$

The structure constraint is obtained via the inverse function of `Lval2Prog` (`Lval2Prog` is a bijective function [38]).

Once a repair condition is obtained through Algorithm 2, we feed this repair condition to a Partial MAX-SMT solver. If the solver finds a model, this model can be used to construct an expression using `Lval2Prog` introduced in Section 2.4.1. Note that a Partial MAX-SMT solver preserves as many original connections as possible, which guarantees that DirectFix changes the minimal number of program expressions, as formally described below.

Definition 8 (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.*

The use of soft constraints reduces synthesis time. Our experiments demonstrate that a Partial MAX-SMT solver implemented on top of an SMT solver can find a solution for a formula with soft constraints for some of the

ALGORITHM 2: Repair condition generation

```
   Input: trace formula $\phi_p$ and test suite $TS$
   Output: repair condition as a pair of hard and soft constraints
1  $Hard, Soft := True, True$;
2  $Expr := \{e \mid e$ is a rvalue expression of $\phi_p\}$;
3  foreach test case $t \in TS$ do
4      foreach $e \in Expr$ do
5          $v :=$ a fresh variable;
6          $C' :=$ select additional component for $e$;
7          $\phi_p := \phi_p[e \mapsto v]$;
8          $\phi_{\text{encoding}} :=$ cmpnt for components of $e$ and components $C'$;
9          $\phi_{\text{struct}} :=$ structural constraints for $v = e$;
10         $Hard := Hard \wedge \phi_{\text{encoding}}$;
11         $Soft := Soft \wedge \phi_{\text{struct}}$;
12     $Hard :=$ Rename($Hard \wedge \phi_p$, $t$);
13     $Soft :=$ Rename($Soft$, $t$);
14 return $Hard$, $Soft$;
```

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.

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.

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 $[s_{j-1}, s_j)$. Figure 3.6 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.

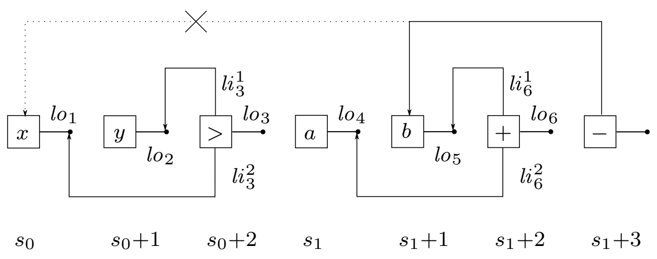


Figure 3.6: Allocating components of the expressions $x > y$ and $a + b$ and the additional component "$-$" on the same interval using floating separators.

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:

$$\begin{aligned}\phi_{\text{range}} := &\bigwedge_{\substack{i\in[1,C]\\(l,r)=scope(i)\\j\in[1..N_i]}} (s_l \le lo_i < s_r \wedge s_l \le li_i^j < s_r)\\ &\wedge \bigwedge_{l<m<r}\left((s_m \le lo_i \wedge \bigwedge_{j\in[1..N_i]} s_m \le li_i^j)\right.\\ &\left.\vee\ (lo_i < s_m \wedge \bigwedge_{j\in[1..N_i]} li_i^j < s_m)\right)\end{aligned}$$

where function *scope* maps a component with the index i to an interval representing the range of program expressions where this component 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.

Type-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 [57]. 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 ex-

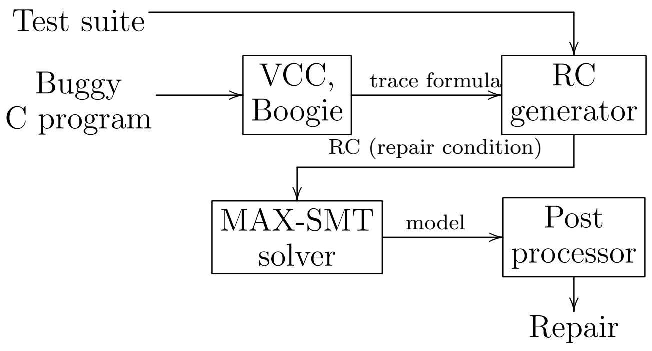


Figure 3.7: DirectFix workflow

ample, 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.

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.

Table 3.1: DirectFix 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

3.4 Implementation

We implement the repair methodology described earlier into a prototype tool, DirectFix. The overall workflow of DirectFix is shown in Figure 3.7. To obtain a trace formula from a buggy program, we use two third-party tools, VCC [18] and Boogie [6]. 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.

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 2. Subsequently, the generated repair condition is fed into our Partial MAX-SMT solver we implemented on top of Z3 [21]. Our MAX-SMT implementation is the core-guided algorithm of Fu and Malik [31].

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

3.5 Evaluation

In this section we present the experimental evaluation of DirectFix. We also compare the repairs generated by DirectFix with those generated by SemFix [73]. We ran our experiments on Intel Core i7-2600 CPU with Ubuntu 12.04 64-bit operating system. Table 3.1 shows our subject programs comprised of eighty nine buggy versions of four subject programs from SIR (Software-artifact Infrastructure Repository) [24] (the number of versions for each subject is shown in the #Versions column) and nine buggy versions of Coreutils reported by Cadar et al [12]. These subjects are the same as the ones used in the SemFix paper [73]. 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).

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 3.2 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

Table 3.2: DirectFix 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

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 3.2 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

Table 3.3: Comparison of DirectFix with SemFix

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%

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.

The Diff column of Table 3.2 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.

```
bool Non_Crossing_Biased_Climb() {
  ...
  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) DirectFix repair (identical with the ground truth repair).

```
bool Non_Crossing_Biased_Climb() {
  ...
  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) SemFix repair.

Figure 3.8: Comparison of repairs for Tcas version 10

Quantitative comparison with SemFix We compare our repairs with those of SemFix [73]. Similar to DirectFix, SemFix also searches for repairs by analyzing the logical semantics of a program, and uses component-based synthesis to generate repairs. 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 3.3 compares the repairs that could be generated by both DirectFix and SemFix. In this table, E stands for Equivalent, S stands for Same Loc, D stands for Diff, and R stands for Regression. 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 3.3, 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.

Qualitative comparison with SemFix Lastly, we provide a couple of concrete examples of repairs generated by DirectFix and SemFix. Figure 3.8b 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 truth repair. Meanwhile, Figure 3.9 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.

```
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) 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) SemFix repair.

Figure 3.9: Comparison of repairs for Replace version 26

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

Chapter 4

Scalable multiline patch synthesis

In the case of the semantic program repair technique, low scalability has been the main source of criticism, despite its promising results in terms of the number of generated patches [73] and the high quality of repairs [69]. We show in this chapter how the semantics-based repair methodology can also scale up to the same level as the most advanced search-based repair tools such as SPR and GenProg. Semantics based repair methods often work by extracting a *repair constraint* typically via symbolic execution. This repair constraint acts as a specification to guide program synthesis - so a patch satisfying the repair constraint can be synthesized. The key enabler for scalable multi-line bug fix in this chapter, is our novel lightweight repair constraint that we call an *angelic forest*. This angelic forest is automatically extracted via symbolic execution. As compared to the repair constraints used in the previous work [73, 69], the angelic forest is simpler, and its size is *independent* of the size of the program under repair, thereby making our repair method scale. Our angelic forest, despite its simplicity, contains enough semantic information to enable multi-location bug fix. Among existing search-based repair tools, SPR does not support multi-line fixes. While GenProg [56] can change multiple locations of the program, a recent study on GenProg repairs [85] shows that seemingly complex repairs generated from GenProg are in the overwhelming majority of cases in fact functionally equivalent to single line modification.

When evaluated with the largest of the GenProg ICSE2012 subjects, our open-source repair tool Angelix successfully generated repairs including on wireshark and php subjects. The number of repairs generated by Angelix (28) is larger than in GenProg (11), and also generally comparable to SPR (31). While in one subject (libtiff), Angelix generated more repairs than SPR, and in another subject (php), SPR generated more repairs. In the remaining 3 subjects, both tools produced the same number of repairs.

More importantly, we note that even though a recent work [85] points to functionality deleting repairs by GenProg, the SPR tool [62] (which was produced by the same authors) itself was found to generate many functionality deleting repairs, because it generates many trivial branch conditions. Such trivial branch conditions (conditions which are always true or always false) introduce functionality deletion; *e.g.*, consider the following SPR repair for a libtiff defect where the shaded part is the fix inserted by SPR.

```
if (td->td_nstrips > 1
    && td->td_compression == COMPRESSION_NONE
    && td->td_stripbytecount[0] != td->td_stripbytecount[1]
    && !(1))
```

We found that in SPR the overall functionality-deleting repair rate across the GenProg benchmark subjects is 45%. In fact, in the `libtiff` subject, the percentage of functionality deleting repairs in the SPR tool [62] goes up to an alarming 80% !! In contrast, our semantic analysis based repair tool produced functionality-deleting repairs significantly less frequently when the same tests were used (21%). For the aforementioned defect, Angelix synthesizes a patch that is identical with the developer-provided patch (shown in Figure 4.2b), which does not delete functionality. Furthermore, the repairs generated by Angelix include five multi-location bugs which have not been fixed by the existing tools. Last but not the least, we report that the well-known Heartbleed vulnerability[1] was automatically fixed by our tool, generating a repair that is smiliar to the developer-provided patch. To the best of our knowledge, ours is the first work that reports the automated repair on Heartbleed. Overall, we present a semantic analysis based program repair method which balances the requirements of scalability (repairing large programs), repairability (repairing a large number of defects) and patch quality (changing the functionality of the program in a way developers would agree with, instead of simply deleting functionality).

4.1 Motivating example

Figure 4.1 shows code changes made to fix coreutils bug 13627. In the buggy version, the call of `xzalloc` (line 4), which allocates a block of memory, causes a segmentation fault. A fix involves adding an if conditional before the prob-

[1]Heartbleed bug: `http://heartbleed.com`

```
- if (max_range_endpoint < eol_range_start)
-   max_range_endpoint = eol_range_start;

- printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
+ if (max_range_endpoint)
+   printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
```

(a) The developer-provided bug patch for coreutils bug 13627 where multiple locations are repaired

```
if (max_range_endpoint < eol_range_start)
   max_range_endpoint = eol_range_start;

if (1)
   printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
```

(b) The buggy version after semantics-preserving transformation (the shaded part is added)

```
if (α)
   max_range_endpoint = β;

if (γ)
   printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
```

(c) Suspicious expressions are replaced with symbolic variables

```
if (0)
   max_range_endpoint = eol_range_start;

if (! (max_range_endpoint == 0))
   printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
```

(d) A repair generated from our repair algorithm; expressions in the shaded areas are synthesized from our repair tool, Angelix.

Figure 4.1: Angelix motivating example

lematic call to `xzalloc` (line 5). Only when variable `max_range_endpoint` has a non-zero value, `xzalloc` can be called in the fixed version. In addition to adding an if conditional, the fix requires removing an existing if statement (lines 1–2). Without this removal, `max_range_endpoint` is overwritten with a non-zero value of `eol_range_start` (line 2), and as a result, the new if conditional `if (max_range_endpoint)` cannot successfully prevent the problematic call to `xzalloc`.

This simple example demonstrates the complexity of multi-line repairs fixing multiple buggy locations. The key difficulty is that a change made in one location can also change the remaining program execution that should proceed to be repaired. More conceptually speaking, the fix space of a given buggy program keeps changing along with the program change made at each buggy location. The state-of-the-art search-based repair algorithm such as SPR [62] (also known as the generate-and-validate methodology) is currently restricted to fixing a single location. It is unclear, as stated in [62], how a search-based repair algorithm can be extended to fix multiple-location bugs such as the one shown in Figure 4.1a, while maintaining its efficiency. Meanwhile, among the state of the art of semantics-based repair methodology such as SemFix [73] and DirectFix [69], DirectFix already supports multiple-location fix. Essentially, DirectFix maintains all semantic information of the program (in the form of a logical formula), and this makes it possible to keep track of how the fix space changes. Thus SemFix is more scalable and applies one line fixes, while DirectFix is less scalable but can produce multi-line fixes.

In this chapter, we discuss how semantics-based repair can scale, while preserving its ability to repair multiple locations. Figures 4.1b–4.1d show at a high level how our repair algorithm generates a repair from our running example. The generated repair shown in Figure 4.1d is functionally equivalent to the developer-provided repair, despite its cosmetic differences. The first piece of a repair, `if (0)` in line 1, makes the statement in line 2 skipped over, which is functionally equivalent to removing the corresponding statement. The second piece of a repair, `if (!(max_range_endpoint == 0))`, is also functionally equivalent to the developer-provided repair, `if (max_range_endpoint)`. Here, the cosmetic difference is merely due to our current implementation of the component-based synthesis algorithm we use to synthesize a repair.

Our repair algorithm starts from transforming the original buggy program into a functionally equivalent one shown in Figure 4.1b where we add an if conditional, `if (1)`, before each unguarded assignment statement (this is heuristics we currently use). Afterwards, our repair algorithm replaces user-configured n most suspicious expressions—chosen based on the result of statistical fault localization—with symbolic variables, as shown in Figure 4.1c where conditional expressions and the right-hand side of an assignment are replaced with symbolic variables. The user of our repair algorithm can configure the number and kinds of suspicious expressions that can be made *symbolic*; such expressions include conditional expressions, right-hand sides of assignments, and function parameters. Our repair algorithm proceeds to run symbolic execution over the program in Figure 4.1c with provided tests to collect the semantic information necessary to repair the given buggy program. Using this extracted semantic information, we synthesize repair expressions. To synthesize a repair, we use component-based patch synthesis algorithm based on MAX-SMT as in Chapter 3. This results in a repair close to the original program, because the structures of the original buggy expressions are maximally preserved. The resultant small patches can bring in various benefits such as improved maintainability of patches (simple patches are easier to understand than complex patches), and reduced risk for regression (simple patches are less likely to change the correct behavior than complex patches).

4.1.1 Concise semantic signature for repair

In order to synthesize a repair, our repair algorithm collects the following pieces of semantic information of the program. First, we need to know whether for each test, there exists a program path through which a given test passes. Our repair algorithm detects such test-passing paths via controlled symbolic execution — controlled in the sense that we control which execution paths are explored during symbolic execution by installing symbolic variables (in our example, α, β, and γ). In our running example, given a program of Figure 4.1c, symbolic execution explores different paths at the if conditionals in line 1 (if (α)) and line 4 (if (γ)). If a test-passing path is not detected, we make the next (user-configured) n suspicious expressions as symbolic, and repeat the procedure to find test-passing paths. On the other side, the existence of a test-passing path π that goes through the installed symbols implies the existence of a concrete value for each symbol that makes the test pass. As the second piece of semantic information, we infer these values (called *angelic values*) using a constraint solver. Lastly, we need to know the program state (called *angelic state*) at each installed symbol in the test-passing path. For example, in order to synthesize a repair expression, `!(max_range_endpoint == 0)`, at line 4 of Figure 4.1d, the value of the variable `max_range_endpoint` should be known. Our repair algorithm collects the values of the visible program variables at each symbol-installed program location. These variables are used as synthesis ingredients when synthesizing repair expressions. The following shows the semantic signature of our running example when two tests (t_1 and t_2) are provided.

$$
\begin{aligned}
t_1 : \{\ & \pi_1 : \langle(\alpha, False, \sigma_1), (\gamma, False, \sigma_2)\rangle, \\
& \pi_2 : \langle(\alpha, True, \sigma_3), (\beta, 0, \sigma_4), (\gamma, False, \sigma_5)\rangle\ \} \\
t_2 : \{\ & \pi_3 : \langle(\alpha, False, \sigma_6), (\gamma, True, \sigma_7)\rangle, \\
& \pi_4 : \langle(\alpha, True, \sigma_8), (\beta, 3, \sigma_9), (\gamma, True, \sigma_{10})\rangle\ \},
\end{aligned}
$$

where t_i referes to a test, π_i denotes a test-passing path, and $\sigma_i : Variables \rightarrow Values$ denotes an angelic state.

The preceding semantic signature—which we call an angelic forest as defined in Definition 11—concisely captures all three pieces of semantic information we need to synthesize a repair. First, the fact that there exist two execution paths (π_1 and π_2) that make test t_1 pass is encoded in $t_1 : \{\ \pi_1, \pi_2\ \}$. Similarly, test t_2 can also pass in two execution paths, π_3 and π_4. Note that the suggested repair shown in Figure 4.1d follows path π_1 in test t_1, and π_3 in t_2. Second, the concrete value of each symbol is denoted at each test-passing path. For example, in path π_1, symbol α and γ should have value False, as denoted with $\pi_1 : \langle(\alpha, False), (\gamma, False)\rangle$. The concrete value of symbol β does not appear because statement max_range_endpoint = β of Figure 4.1c is not executed in path π_1. Meanwhile, in path π_2, the values of all three symbols appear as denoted with $\pi_2 : \langle(\alpha, True), (\beta, 0), (\gamma, False)\rangle$. Lastly, angelic state σ_i informs about the values of variables to use in repair synthesis. The same variable can have different values along a path, and that is why each instance of a symbol is associated with its own angelic state. In our coreutils example, σ_2(max_range_endpoint) is zero, and this is why the suggested repair expression, `!(max_range_endpoint == 0)`, returns the concrete value of γ, False, as specified in π_1.

4.1.2 Reasons for scalability

As will be shown in the experimental results (Section 4.3), our repair method can handle programs as large as Wireshark (2814 KLoC) and generate multi-location repairs. There are multiple reasons why our repair method scales. First, we use a *lightweight semantic signature* for program synthesis. Compare our semantic signature with the one used in DirectFix [69] which can also synthesize multi-location repairs. The semantic signature used in DirectFix is essentially the semantics of the whole program. There, the relationship between each and every expression appearing in the program is maintained, unlike in our new semantic signature. As a result, the semantic signature of DirectFix becomes more lengthy and complex, as the size of the program increases. It it important to note that the semantic signature is the specification for repair synthesis in the sense that a synthesized repair should respect the provided semantic signature, as explained with our running example. Our lightweight semantic signature reduces the burden of the repair synthesizer, resulting in more efficient repair synthesis.

Second, our repair algorithm performs *controlled symbolic execution* with a few selected suspicious expressions, instead of usual symbolic input. Using this controlled symbolic execution, we explore only a restricted number of feasible execution paths involving only the selected few suspicious expressions. Also, we initially perform symbolic execution only with a subset of the provided test-suite to reduce the running time of symbolic execution. Only when some of remaining tests fail with the synthesized repair, we perform additional symbolic execution with these failing tests.

Lastly, our repair algorithm initiates repair synthesis only when there exists an angelic forest—the semantic signature for repair. The absence of an angelic forest for a chosen n suspicious locations implies that it is not possible to repair the bug by changing these n locations. Symbolic execution finds an angelic forest (or proves the absence of an angelic forest) efficiently by exploring only feasible execution paths. Our repair algorithm does not waste the resources to synthesize a repair if there is no angelic forest.

4.2 Methodology

Our repair methodology consists of the following 4 steps: (1) program transformation, (2) fault localization, (3) extracting a repair constraint, and (4) patch synthesis. In the first step, we perform semantics-preserving program transformation to expand the defect class our repair algorithm can fix. For example, we showed in Section 4.1 that `if (1)` can be added before each unguarded statement. More generally, our repair framework is transparent to the addition of more semantics-preserving program transformation schemas. In the second step, we perform statistical fault localization. We use the Jaccard formula [16], considered most effective for automated program repair according to [84]. Since our repair algorithm modifies buggy expressions, we apply the Jaccard formula at the expression level, instead of at the statement level. The last two steps distinguish semantics-based repair methods from search-based repair methods such as GenProg and SPR. Semantics-based methods extract a repair constraint from the program under repair typically via symbolic execution. This repair constraint acts as a specification to guide program synthesis—so a patch satisfying the repair constraint can be synthesized.

The key novelty of our repair method is our new lightweight repair constraint that we call an *angelic forest*. The size of this angelic forest is *independent* of the size of the program under repair. This is the main reason why our new repair method can scale. Our angelic forest, despite its simplicity, contains enough semantic information to enable multi-location bug fix. In the following, we formally define our angelic forest (Definition 11) based on the definition of an angelic value (Definition 9) and an angelic path (Definition 10).

Definition 9 (Angelic value). *Let p be a program, t be a failing test, e be a program expression and e^i be its i-th instance in the execution trace of t.* Angelic value *c is such that replacing expressions e^i with c during the execution of t makes p pass the test t.*

Definition 10 (Angelic path). *Let p be a program, E be a set of program expressions in p, t be a test. An* angelic path *is a set of triples (e^i, c, σ) where e^i is the i-th instance of an expression $e \in E$ appearing in the execution trace of t, c is an angelic value of e^i, and σ represents an angelic state at e^i, such that replacing all e^i in the execution trace of t with the corresponding angelic* *values c forces (1) the program p to pass the test t and (2) the program p to be in the state σ when the expression e^i is evaluated.*

Definition 11 (Angelic forest). *Let p be a program, E be a set of program expressions in p, t be a test.* Angelic forest A_t *for the test t is a set of angelic paths for t.*

We extract an angelic forest via *controlled* custom symbolic execution — controlled in the sense that instead of initiating symbolic execution with symbolic input, we install symbols at a few suspicious program locations — chosen based on a statistical fault localization result — to control the execution paths to be explored during symbolic execution.

Let p be a program, e be a program expression (this naturally generalizes to a set of expressions E), $(\sigma_{in}, \sigma_{out})$ be a test. Angelix first extracts a set of triples $\{(\pi, \theta_c, \theta_{out})\}$ such that $\{(\pi, \theta_{out})\} = \mathit{SymExec}(p', \sigma_{in})$, where $p' := p[e \mapsto \texttt{choose}()]$, $\texttt{choose}()$ is a function that returns a fresh symbolic variable α_i each time it is executed. For each path π, θ_c indicates a symbolic state in the context of which the function $\texttt{choose}()$ is called when the program is symbolically executed along π. Then, Angelix executes Algorithm 3 to extract an angelic forest. This algorithm takes in the test $(\sigma_{in}, \sigma_{out})$ and the set of pairs extracted through controlled symbolic execution. For each explored path, given the expected output σ_{out} available in the test, we find a model of $\pi \wedge \theta_{out} = \sigma_{out}$ via a constraint solver. This model is used to extract an angelic path, and thereafter grow the angelic forest. We implement our custom symbolic execution on top of KLEE [12].

Given an angelic forest, Angelix can construct an expression from a given

ALGORITHM 3: Angelic forest extraction

```
Data: test $(\sigma_{in}, \sigma_{out})$, set of triples $\{ (\pi, \theta_c, \theta_{out}) \}$
Result: angelic forest $A_t$
$A_t := \emptyset$;
foreach $(\pi, \theta_c, \theta_{out})$ do
    $\psi := \pi \wedge \theta_{out} = \sigma_{out}$;
    if $isSAT(\psi)$ then
        $\{\alpha \mapsto n_1, ...\} :=$ getSatisfyingAssignment($\psi$);
        $c := n_2$;
        $\sigma_c := [\![\theta_c]\!]_{\{\alpha \mapsto n_1, ...\}}$;
        $A_t := A_t \cup \{ (e, c, \sigma_c) \}$;
    end
end
return $A_t$;
```

library of components that satisfies a given angelic forest. A synthesized repair, when executed, follows one of the angelic paths for each test, thereby all tests pass. In these angelic paths, each repaired expression returns its corresponding angelic value specified in the corresponding angelic path. More formally, for a given angelic forest A_t and a set of components c_1, ..., c_n, it produces an expression e constructed from c_1, ..., c_n that by solving the following second-order formula:

$$\exists e \in L_\Gamma . \bigvee_{path \in A_t} \bigwedge_{(e^i, c, \sigma) \in path} [\![e]\!]_\sigma = c$$

Such e takes the angelic value c for each angelic state σ and therefore passes the test t by construction.

In order to solve the above formula, we apply the repair algorithm described in Chapter 3. Specifically, Angelix finds a patch requiring minimal changes by using MAX-SMT solver. The ability to maximally preserve the original source code is important for two reasons. First, our hypothesis is that such a minimal patch would be preferred by developers. Minimal patches are easier to validate and they are less likely to change the correct behavior of the original program than more complex patches as demonstrated in [69]. Second, when synthesizing a repair for multiple suspicious expression, MAX-SMT-based repair serves as fault localization, that is the repair algorithm simultaneously identifies which expressions to modify and how to modify them. Without this property, synthesizer would always modify all the suspicious expressions making milti-location repair not practical due to the complexity of patches. As will be shown in Section 4.3, this way of synthesis provides higher-quality repairs than SPR.

Optimization To control the number of symbolic execution sessions, we use the following iterative approach. First, we start from a small subset of the test suite that provides the highest coverage of the suspicious locations. Then, we infer angelic forest for this reduced test suite and synthesize a patch. If the generated patch causes a regression in the whole test suite, we add the counter-example test to the test suite. We repeat these steps until all test cases become passing. Regarding running time, there is one more advantage of semantics-based methods. Contrasting to search-based methods where the software under repair is rebuilt and retested frequently due to a high number of repair trails, our semantics-based method finds a repair in one or a small number of trials, and the cost for rebuilding and retesting is significantly smaller. Because of type coercion and the absence of a separate boolean type in C programming language, it is difficult to distinguish between types of program expressions. On the other side, knowing precise types increases the probability of synthesizing correct repair as well improves the synthesis performance. For this reason, we analyze the usage of suspicious expression and visible variables to collect type constraints. Then, these type constraints are used to infer more precise types for program expressions and variables. As an example, we assign a boolean type to the expressions used as if conditions.

Soundness and completeness While the size of an angelic forest independent of the size of the program, it also under-approximates the fix space — that is, it cannot capture whole (possibly infinite) set of values for the suspicious expressions that make the test pass. Our repair method based on an angelic forest is sound in the sense that the repair obtained by our repair method indeed passes all the provided tests. However, our repair method is incomplete in the sense that it may not produce some repairs, due to the under-approximation of angelic values used in an angelic forest, that can otherwise be synthesized.

4.3 Evaluation

We evaluate our repair method to answer the following two research questions.

RQ1. Can our repair method generate repairs from large-scale real-world software?

RQ2. Can our repair method fix multi-location bugs?

Table 4.1: Angelix experimental results

Subject	LoC	Tests	Versions	W/I Our Defect Class	Fixed Defects				Equiv. to Developer Fixes				Time (min)
					Angelix	SPR	GenProg	AE	Angelix	SPR	GenProg	AE	
wireshark	2814K	63	7	4	4	4	1	4	0	0	0	0	23
php	1046K	8471	44	12	10	18	5	7	4	8	1	2	62
gzip	491K	12	5	2	2	2	1	2	1	1	0	0	4
gmp	145K	146	2	2	2	2	1	1	2	1	0	0	14
libtiff	77K	78	24	12	10	5	3	5	3	1	0	0	14
Overall			82	32	28	31	11	19	10	11	1	2	32

4.3.1 Evaluation setup

Subjects The first 4 columns of Table 4.1 show our subject programs, the size of each program in LoC, and the number of tests and buggy versions of each subject (in the Tests and Versions columns, respectively). Our subjects are taken from the GenProg ICSE2012 benchmark [56]. These subjects have been also used in the literature to evaluate other repair tools such as GenProg and SPR [62]. In particular, wireshark and php are among the largest subjects in the benchmark. We use these large subjects to evaluate the scalability of our repair method. We omit three subjects of the benchmark (python, lighttpd, and fbc) because we could not run these subjects on KLEE [12]. KLEE currently cannot support all library functions. Note that this limitation of KLEE is orthogonal to our repair approach. We use the same subjects to evaluate our second research question, i.e., multi-location repairability. Besides, in addition to these subjects in the GenProg benchmark, we add 3 multi-location bugs extracted from CoREBench [10] to our subject list. We add these multi-location bugs because the GenProg benchmark does not have many multi-location bugs in the fix space of our tool.

Tests and correctness of patches The Tests column of Table 4.1 shows the number of tests of each subject in the GenProg benchmark. We rectified the original test scripts delivered in the GenProg benchmark to address the problems pointed out in [85] such as the weak proxy problem. Meanwhile, each coreutils version available in CoREBench contains a failing test that can reproduce the defect. We use these failing tests and the existing tests available in the subject. All the repairs generated from our tool are manually inspected for its correctness. We consider a repair correct only if the generated patch is functionally equivalent to the developer-provided patch.

Configurations Our repair tool allows to control the following parameters of our repair algorithm — the maximum number of suspicious locations that can be repaired at the same time, the kinds of suspicious expressions, and the kinds of (semantics-preserving) program transformation. First, for the maximum number of suspicious locations, we used the value between 1 and 10 (inclusive). Second, for the kinds of suspicious expressions, we used the following three levels. A higher level is more inclusive. At the lowest level, we allow only conditional expressions to be considered suspicious. At the next level, we also consider the right-hand side expressions of assignment statements. At the highest level, we also consider function parameters. At all levels, only side-effect/function-call free expressions are considered. Lastly, for the semantics-preserving program transformation, we allow to add `if (1)` before each unguarded statement. We also allow to add `if (0) break;` at the end of a loop body to be able to produce a repair requiring to break a loop. We provided our tool with the names of buggy source code files (which are known through developer-provided fixes), as in the previous studies [56, 107, 62]. All our experiments were performed on Intel Xeon E5-2660 2.20GHz CPU with Ubuntu 14.04 64-bit operating system. We used 12 hours as the timeout of each repair session.

Defect class As pointed out in [72], defining defect classes supported by a repair algorithm helps evaluate the efficacy of a repair algorithm (how effectively bugs in the target defect class can be repaired), and compare multiple repair algorithms one another (which repair algorithm generates repairs more effectively for the target defect class). The defect class of our repair algorithm can easily be defined in terms of the fix that can be synthesized. Our repair tool can synthesize side-effect/function-call free expressions that can be composed of boolean/arithmetic/relational operators, variables available, and constants. Also, by using semantics-preserving program transformation (i.e., adding `if (1)` before unguarded statements), fixes requiring statement deletion is effectively included in our defect class, as shown with the motivating example in Section 4.1. However, our repair tool currently cannot add a new statement/variable. The W/I Our Defect Class column of Table 4.1 shows the number of defects of each subject that are in our defect class. We manually inspected each developer-provided fix to check whether the corresponding defect is in our defect class or not. Although there can be other possible fixes different from a developer-provided fix, it is infeasible to consider all unknown possible fixes. Thus, we additionally only inspected fixes from other repair tools (SPR, GenProg, and AE [107]) and ours. The number of defects within our defect class is less than the number of buggy versions (shown in the Versions column), because some bug fixes in the benchmark require adding new statements/variables.

4.3.2 Results from GenProg benchmark

Table 4.1 shows our results from the GenProg benchmark. The first five columns are already explained earlier, and self-explanatory. We only mention that subjects of the table are sorted by their sizes. The Fixed Defects column shows the number of fixed defects by our tool, Angelix, and other tools — SPR, GenProg, and AE. Similarly, the Equiv. to Developer Fixes column shows the number of fixes functionally-equivalent to the developer-provided fixes out of the fixed defects. The results from other tools (SPR, GenProg and AE) are taken from [62]. Lastly, Time column shows the average running time of our tool for each subject, when repairs were found. The running time of the other tools are available in their respective papers [56, 62, 107], although each tool is experimented on a different type of machine. In all subjects with different sizes between 77 KLoC and 28214 KLoC, our tool successfully generated repairs for some defects. Our tool generated repairs for most defects in our defect class (28 out of 32), and more than third of these repairs (10 out of 28) are functionally equivalent to developer-provided repairs. Three defects in our defect class were not fixed due to imprecise statistical fault localization (e.g., buggy initialization of a global variable was not ranked high). One remaining defect requires modifying a string value (a character sequence) in a way that cannot be handled by our current solver (the length of the string should change in a fix). As shown in the Time

Table 4.2: Number of defects exclusively repaired by each repair tool across the subjects

Subject	Angelix	SPR	GenProg	AE
wireshark	0	0	0	0
php	0	4	0	0
gzip	1	0	0	0
gmp	0	0	0	0
libtiff	5	0	0	0
Overall	6	4	0	0

column, the average running time of our repair tool is about half an hour, when a repair is found.

Angelix, an implementation of our new semantics-based repair algorithm, successfully generates repairs from 5 real-world software as large as 77–28214 KLoC in 32 minutes on average. This result shows that a semantics-based repair can scale.

Angelix fixed 2 multi-location bugs of the GenProg benchmark. We show these results along with the results from the multi-location bugs of coreutils in Section 4.3.3.

Comparison with other repair tools

When compared with the state-of-the-art repair tool, SPR, our tool shows higher repairability (more defects are repaired in our tool) in libtiff (10 vs 5), and lower repairability in php (10 vs 18). In the remaining 3 subjects, both tools shows the same repairability. This varying repairability across

```
if (td->td_nstrips > 1
    && td->td_compression == COMPRESSION_NONE
    && td->td_stripbytecount[0] != td->td_stripbytecount[1])
```

(a) The buggy location of libtiff-d13be72c-ccadf48a

```
if (td->td_nstrips > 2
    && td->td_compression == COMPRESSION_NONE
    && td->td_stripbytecount[0] != td->td_stripbytecount[1])
```

(b) The repair generated by our tool, Angelix

```
if (td->td_nstrips > 1
    && td->td_compression == COMPRESSION_NONE
    && td->td_stripbytecount[0] != td->td_stripbytecount[1]
    && !(1))
```

(c) The repair generated by SPR

Figure 4.2: Comparison of repairs from Angelix and SPR

the subjects is related to the different defect classes of Angelix and SPR. For example, the defect class of SPR contains inserting a function call such as `memset`, and 5 php defects are included in this defect class. Meanwhile, Angelix can fix multiple buggy locations, and two libtiff multi-location defects are exclusively fixed by our tool. More generally, Table 4.2 shows the number of defects exclusively repaired by each repair tool across the subjects. Our tool produced the most number of unique repairs, as compared to SPR, GenProg and AE.

We also qualitatively compare the repairs from Angelix and SPR. Figure 4.2 shows a buggy location of libtiff-d13be72c-ccadf48a in (a), the repair generated by our tool in (b), and the repair generated by SPR in (c). The difference between the original code and each repair is shaded. The SPR repair looks problematic, because it simply deletes functionality by disabling

Table 4.3: The number of functionality-deleting repairs

Subject	Angelix			SPR		
	Fixes	Del	Per	Fixes	Del	Per
wireshark	4	1	25%	4	1	25%
php	10	3	30%	18	7	39%
gzip	2	0	0%	2	1	50%
gmp	2	0	0%	2	0	0%
libtiff	10	2	20%	5	4	80%
Overall	28	6	21%	31	13	42%

the block of code in the then branch. Indeed, this patch is not functionally equivalent to the developer-provided patch. Still, such an overfitting repair [95] (an incorrect repair that merely passes the provided tests) can be helpful in debugging, because the user can at least see that the (incorrectly) repaired if conditional may be buggy. However, compare this SPR repair to the repair generated by our tool shown in Figure 4.2b. Our repair spots the buggy location more precisely down to `td->td\_nstrips > 1`. This is because our repair tool generates a repair that is close to the original buggy expression by using a MAX-SMT solver. As a result, a problematic buggy location can be pinned down more precisely. In fact, our repair is identical with the developer-provided repair in this case.

Incorrect repairs that merely delete functionality are common in SPR repairs. Table 4.3 compares the number of repairs that delete functionality between our tool and SPR. In each tool (the Angelix and SPR column, respectively), we list the number of fixes generated in each tool (the Fixes

Table 4.4: Experimental results for multi-location defects.

Defect	Fixed Expressions
libtiff-4a24508-cc79c2b	2
libtiff-829d8c4-036d7bb	2
coreutils-00743a1f-ec48bead	3
coreutils-1dd8a331-d461bfd2	2
coreutils-c5ccf29b-a04ddb8d	3

column), the number of functionality-deleting fixes (the Del column), and the percentage of functionality-deleting fixes out of generated fixes (the Per column). In five subjects used in our experiments, 42% of SPR-generated repairs delete functionality, and in the libtiff subject, the percentage goes up to 80%. Even if the three omitted subjects (python, lighttpd, and fbc) are also considered, the percentage of functionality deleting repairs stays even at a high rate of 45%. GenProg and AE also often generate functionality-deleting repairs, as reported in [85]. In comparison, Angelix generates functionality-deleting repairs less frequently (21%).

Angelix is not only scalable but also less frequently generates functionality-deleting repairs than the existing tools such as SPR and GenProg.

4.3.3 Results from multi-location bugs

Table 4.4 shows the experimental results for multi-location defects of the GenProg benchmark and coreutils. The "Fixed Expressions" column shows the number of expressions fixed by our tool. Angelix produced a repair functionally equivalent to the developer-provided one for coreutils-00743a1f-ec48bead. Meanwhile, in coreutils-1dd8a331-d461bfd2, while two conditional expressions are repaired in a functionally similar way to the developer patches, the output message is not corrected in our repair, because this message is not part of the the oracle in the tests used for repair. In coreutils-c5ccf29b-a04ddb8d, the developer-provided repair uses function calls that are not used in our repair.

We note that the number of defects covered by our multi-location defect class is limited at least in the two benchmarks we investigated (the GenProg benchmark and CoREBench). Many developer-provided fixes for multi-line defects involve adding new variables, statements, and functions. We believe that the research in automatic patching should be developed into such more sophisticated patches, and our multi-location defect class is on the pathway toward such a direction. To the best of our knowledge, only our repair tool can currently generate (non-functionality-deleting) fixes for multi-location bugs in large-scale real-world software.

4.4 Experience with Heartbleed bug

We applied our repair tool to a buggy version of OpenSSL (OpenSSL-1.0.1-beta1) that has the infamous Heartbleed bug. Heartbleed is considered one of the most dangerous in the annals of security vulnerabilities, because attackers can exploit Heartbleed to steal important confidential data, including login cookies, passwords, and private cryptographic keys, without leaving a trace

```
if (hbtype == TLS1_HB_REQUEST) {
   ...
   memcpy (bp, pl, payload);
   ...
}
```

(a) The buggy part of the Heartbleed-vulnerable OpenSSL

```
if (hbtype == TLS1_HB_REQUEST
   && payload + 18 < s->s3->rrec.length) {
   /* receiver side: replies with TLS1_HB_RESPONSE */
}
```

(b) A fix generated by our tool, Angelix

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0;
...
if (hbtype == TLS1_HB_REQUEST) {
   /* receiver side: replies with TLS1_HB_RESPONSE */
}
else if (hbtype == TLS1_HB_RESPONSE) {
   /* sender side */
}
return 0;
```

(c) The developer-provided repair

Figure 4.3: Heartbleed bug and its fixes

from numerous servers depending on OpenSSL to run their services. We report that we could automatically fix the Heartbleed bug using our repair tool. To the best of our knowledge, this is the first work that reports the automated repair on Heartbleed.

The Heartbleed bug is an instance of a buffer over-read (CWE-126), one of common weakness of C/C++ programs. Exploiting this weakness, attackers can read beyond the region of a buffer that is originally intended by the programmers. Figure 4.3a shows where this weakness exists in the Heartbleed-vulnerable OpenSSL. The main culprit is `memcpy(bp, pl, payload)` (line 3) where attackers can assign `payload` (the third parameter of `memcpy` that sets the number of bytes to copy from the target memory region) a larger value than the size of buffer `pl`, the target memory region. The programmer of OpenSSL made a (common) mistake of not putting a bounds check before this problematic `memcpy`.

We applied Angelix to OpenSSL for repairing the Heartbleed bug. We obtained publicly available tests[2], and added four more tests to cover missing corner cases. Figure 4.3b shows the fix generated by our tool in the shaded area. With this fix, `memcpy` cannot be invoked if `payload` is larger than is allowed by the TLS/DTLS network protocol (the buggy code of OpenSSL is the implementation of these protocols). Our repair synthesizer could compose this repair with `payload` and `s->s3->rrec.length`, both of which are in the scope at the fixed if conditional (they appear in the other parts of the buggy function). In comparison, the developer-provided repair is shown in the shaded area of Figure 4.3c. In both repairs, the failure of the bounds check, which is performed by the added conditional, makes the receiver simply return zero, instead of replying with a response packet. Based on our experience with Heartbleed, we make the following assessment.

Automated repair techniques, such as Angelix, are powerful enough to fix some of well-known and serious software vulnerabilities like Heartbleed.

[2]Heartbleed tests: `https://code.google.com/p/mike-bland/source/browse/heartbleed/`

Chapter 5

Semantic repair using a reference implementation

The primary reason for the low quality of automatically generated patches is *the lack of specifications* of the intended behavior. Most program repair systems rely on tests as the correctness criteria, because a formal specification is often unavailable in practice. However, since tests is an incomplete specification, generated patches often do not correspond to developer intentions, but merely overfit the tests. In order to increase the quality of automatically generated patches, researchers have proposed such techniques as patch prioritization [63], anti-patterns [101], test generation [110, 114], etc. Although these techniques increase the probability of finding correct patches, they nevertheless do not provide any *correctness guarantees* beyond the tests in a given test suite.

To address the overfitting problem, we propose to automatically infer the missing specification for a buggy program from a correct reference program. A reference program is an alternative realization of the same functionality, which is often available for libraries (e.g. standard library implementations, audio codecs, compression algorithms, parsers, cryptographic algorithms), network protocols [59], commodity software (e.g. GNU Coreutils and Busybox implement the same set of UNIX utilities), in the area of digital signal processing [58], web servers and database management systems. Note that a reference program may have a substantially different implementation (different data structures and algorithms), which distinguishes our approach from repair techniques [100] that employ previous program versions. The use of a reference program enables us to alleviate test overfitting and provides additional correctness guarantees.

Ideally, a generated patch should enforce the equivalence of the patched and the reference programs, which poses two challenges: scalability and applicability. First, a recent work [52] reported that a straightforward combination of an equivalence checking system [51] with counterexample-guided inductive synthesis [3] to synthesize equivalence-enforcing patches is not scalable. Second, real-world implementations of the same functionality rarely follow precisely the same specification, e.g. GNU Coreutils implements a superset of the functionality implemented in Busybox and therefore cannot be directly used for equivalence checking.

To address the above challenges, we introduce a methodology of patch generation based on a reference implementation that integrates the notion of conditional equivalence [46] and the scalable patch generation algorithm described in Chapter 4. We rely on the user insight to provide an *input condition* for patch generation that should (1) include a bug-triggering input and (2) correspond to functionality shared by the buggy and the reference programs. Then, our system automatically generates a patch for the buggy program that enforces *conditional equivalence* of the patched and the reference programs, that is equivalence for all inputs satisfying the provided condition. Although the user is only required to provide an input condition (the *property* being checked is derived automatically from the reference program), this still may be non-trivial for applications that involve a complex execution setup. To tackle this problem, we propose a practical approach of introducing an input condition based on the idea of parameterized tests [104], i.e. the condition is defined by injecting symbolic parameters into existing tests.

5.1 Overview

Our approach takes four inputs: a test suite, a buggy and a reference program, and a user-defined input condition (Figure 5.1).

As the first step, the node *Fault localization* of Figure 5.1 represents the identification of suspicious expressions that might need to get repaired. This is done by applying statistical fault localization [42] based on the given test suite and the buggy program. The suspicious expressions in the buggy program get replaced with symbolic variables, denoted as the instrumented buggy program.

As the second step, the module *Symbolic analysis* of Figure 5.1 contains the symbolic execution of the reference program and the instrumented buggy program using the user-defined input condition as a precondition. The result of each symbolic execution is a set of pairs of resulting path conditions and

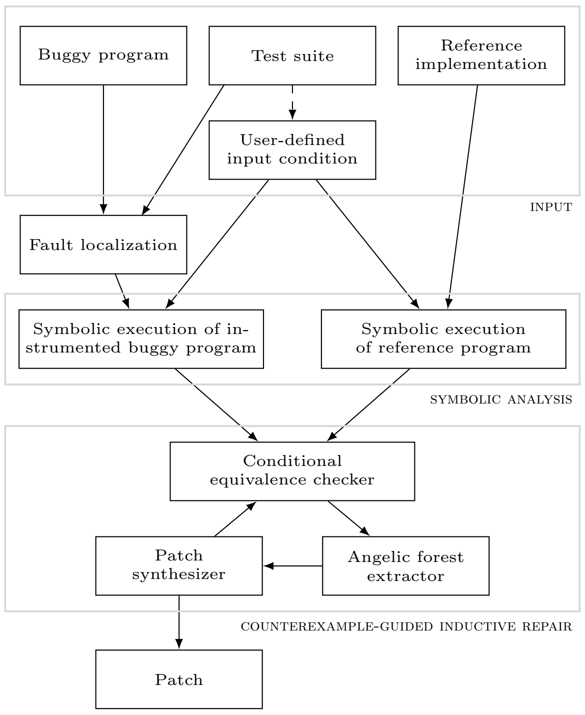

Buggy program
Test suite
Reference implementation
User-defined input condition
INPUT
Fault localization
Symbolic execution of instrumented buggy program
Symbolic execution of reference program
SYMBOLIC ANALYSIS
Conditional equivalence checker
Patch synthesizer
Angelic forest extractor
COUNTEREXAMPLE-GUIDED INDUCTIVE REPAIR
Patch

Figure 5.1: SemGraft workflow.

symbolic output states (see Definition 13) that is used as a specification.

As the last step, the inferred specifications for the reference and the buggy programs are passed into the patch generator that performs a *counterexample-guided inductive repair* loop. Specifically, it performs the following iterations starting from the original buggy expression as the initial (empty) patch:

1. Construct a verification condition (VC) for the patch.

2. Generate a counterexample input that violates the conditional equivalence property by solving VC.

3. Extract an *angelic forest* [70] for the generated input.

4. Synthesize a patch that satisfies the angelic forest.

5. Go to step (1).

This loop repeats until a conditional equivalence-enforcing patch is synthesized or the next patch/angelic forest cannot be found.

To illustrate our approach, we consider the reference program in Figure 5.2a and the buggy program in Figure 5.2c. The reference program implements an algorithm of searching for an element of an array via linear search, while the buggy programs uses binary search. The buggy program contains a bug in line 16.

A crucial element of our approach is the input condition ϕ that has to be defined by the user. The most trivial choice of the input condition would be simply *True*, i.e., checking equivalence for all program input. However, this approach may have a poor scalability as it has been reported in previous works [52]. Instead, we suggest defining the input condition by parameterizing existing tests. Specifically, not all inputs of a test case must be considered concretely, some of them can be handled symbolically. Therefore, the user can transform the test cases in logical constraints and possibly add additional constraints. This represents a practical solution to balance the completeness and scalability of automated program repair. The wider the input condition is formulated, the more complete, in terms of covered input space, will be the generated patch.

To define the input condition ϕ, assume that the user formulates it informally in the following way: we only consider sorted arrays of the length 3 and without duplicates. First, we introduce a mapping between program variables and symbolic variables:

$$\{x \mapsto \gamma, a \mapsto [\alpha_0, \alpha_1, \alpha_2], \mathit{length} \mapsto \delta\}$$

Then, the input constraint is defined as follows:

$$\phi \coloneqq \alpha_0 < \alpha_1 < \alpha_2 \wedge \delta = 3$$

The given test suite contains one negative test case with the input $\{x \mapsto 2, a \mapsto [2, 4, 6], \mathit{length} \mapsto 3\}$ and the expected output 0, because the first element is equal to the searched value 2. The test case passes for the reference program, but fails for the buggy program. The statistical fault localization identifies the expression in line 16 as a suspicious expression, hence, we introduce the symbolic variable β and generate an instrumented buggy program by replacing (`x` $<$ `a[m]`) with β. Note that the test case is not encoded in the input condition ϕ, i.e., the repair steps themselves are independent from any given concrete test input. This test case is only needed for the identification of suspicious expressions.

Assuming ϕ, we execute the reference and instrumented buggy program with a *preconditioned symbolic execution*. Preconditioned symbolic execution (see Definition 12) explores only a subset of program paths that are consistent with the condition ϕ. We will get the results presented in Figure 5.2b and Figure 5.2d. The tables contain so called *specifications* (see Definition 14), which describe the path condition (π^r for the reference program and π^b for the buggy program), the current context for the suspicious expression θ_c, and the output symbolic state (θ^r_{out} for the reference program and θ^b_{out} for the buggy program). The superscript index of β indicates the occurrence id (or instance id) of this expression, since it can be visited more than once during the execution.

With the results of the preconditioned symbolic execution we can build the formula of the verification condition for the considered expression ($\texttt{x} < \texttt{a[m]}$). A verification condition (VC) encodes the following idea: if both executions in the reference and buggy program follow the same path, then their outputs should be the same. The VC will also encode the values of the visible variables in the expression ($\texttt{x} < \texttt{a[m]}$) computed in the symbolic context θ_c that we denote as $\beta = (\texttt{x} < \texttt{a[m]})[\![\theta_c]\!]$. The simplified version of this first iteration VC is presented in Figure 5.3. We skipped contradicting pairs of path conditions from the buggy and the reference program and discarded pairs that contradict the input condition ϕ. Additionally, we simplified the formula by removing lines where the symbolic output states match already, i.e., $\theta^r_{out} = \theta^b_{out}$. In such cases the implication is always $True$ and, hence, it does not provide any additional value. The remaining formula includes the following combinations of paths (represented by the according ids in Figure 5.2b and Figure 5.2d): (r_1, b_3), (r_1, b_4), (r_3, b_6), (r_3, b_7), as also indicated at the beginning of each line in the shown VC.

In order to check the validity of the verification condition, we check the unsatisfiability of its negation as in previous works [51, 79]. The negated VC will be solved using an off-the-shelf SMT solver to generate satisfying values representing counterexamples, which do not satisfy the VC with the current replacement for the suspicious expression. After negating the VC, the SMT solver generates a counterexample input $\{x \mapsto -1, a \mapsto [-1, 0, 1]\}$.

In order to find the correct truth value for β, also called angelic value (see Definition 9), we look at the path condition of the buggy program that leads to the correct output symbolic state, which is here $\theta^r_{out} = 0$ according to the reference program. Comparing with the table in Figure 5.2d, this output can only be reached in the buggy program by following the path b_5. In order to take this specific path with the given input values, β^0 needs to be *False*. This leads to the following angelic forest, which is the input structure for our synthesizer and represents all needed values for the specific suspicious expression (see Definition 11):

$$\{(\beta^0, c, \sigma)\}, \text{ given that } c := \mathit{False}, \sigma := \{x \mapsto -1, a[m] \mapsto 0\}$$

where c represents an angelic value of the considered expression (a value that enables the program to pass the counterexample test) and σ represents an angelic state (the concrete values of program variables in the context in which the expression is executed).

The generated values are used to build the input for a component-based synthesizer, which generates a new patch matching the current synthesizer input. Given this input, the synthesizer returns a plausible patch (`x == a[m]`). After inserting this expression in the VC and negating it, the SMT solver generates a second counterexample input $\{x \mapsto 1, a \mapsto [-1, 0, 1]\}$. The correct output symbolic state for this input is $\theta^r_{out} = 2$ and this matches only the path b_2. In order to take this specific path with the second counterexample,

β^0 needs to be *True*. This leads to an extension of our angelic forest to:

$$\{ \ (\beta^0, \mathit{False}, \{x \mapsto -1, a[m] \mapsto 0)\},$$
$$(\beta^0, \mathit{True}, \{x \mapsto 1, a[m] \mapsto 0\}) \ \}.$$

Given this input, the synthesizer returns the patch (`x` >= `a[m]`). After inserting this expression in the VC and negating it, the SMT returns *unsatisfiable*, i.e., the synthesized patch fulfills all requirements. Note that (`x` >= `a[m]`) is not syntactically equivalent with the correct patch (`x` > `a[m]`), but in this context (i.e. with the preceding if-condition) both expressions are *semantically* equivalent. Our approach results in a patch for the buggy program, so that given the input condition ϕ, the patched program is conditionally equivalent with the reference program.

For comparing with test-driven repair techniques, we applied Angelix [70], which uses a similar path generation algorithm, and hence, it represents the closest existing approach and means a more fair comparison than using any other test-driven repair technique. For our motivating example we observed that Angelix only can produce the plausible patch (`a[m]` < `a[m]`). It fixes only the given negative test case, so in order to generate a correct patch it would be necessary to include more test cases. Since our repair approach is capable of using another program as correctness reference, it generates the input for the synthesizer itself with the presented counterexample-guided approach.

In this motivating example we showed that with our approach it is possible to use a relatively simple reference program (e.g., the linear search) to repair an optimized program (e.g., the binary search). The two programs do not need to be structurally similar, as long as they solve the same problem.

5.2 Methodology

In this section, we formally define three main components of our algorithm: specification inference, verification condition construction and patch synthesis.

5.2.1 Specification inference

The main intuition of our approach is that it is possible to infer a correct specification from a reference implementation and synthesize a patch that enforces this specification in a given buggy program. Hereinafter, we refer to the given reference implementation as the program p_r and the given buggy program as the program p_b.

In order to infer a specification, we use preconditioned symbolic execution defined as follows:

Definition 12 (Preconditioned symbolic execution)**.** *A* preconditioned symbolic execution *procedure PSymExec* : $\mathcal{P} \times \Theta \times \mathcal{L} \rightarrow 2^{\mathcal{L} \times \Theta}$ *is a symbolic execution, in which each path condition is conjoined with a given formula. It can be defined as* $\mathit{PSymExec}(p, \theta, \phi) := \mathit{SymExec}(p', \theta)$*, where* $p' :=$ `assume` $\phi;\ p$.

The implementation of preconditioned symbolic execution is discussed in Section 5.3. As a result of adding the condition ϕ, preconditioned symbolic execution is significantly more efficient than conventional symbolic execution, since it explores only a subset of program paths that is consistent with the condition ϕ.

For a given reference program and an input condition, we infer a symbolic summary of the program computed through preconditioned symbolic execution:

Definition 13 (Symbolic summary)**.** *Let p be a program, ϕ be an input condition, θ is a symbolic state. A* symbolic summary *is a set of pairs $\mathit{Sum}(p, \theta, \phi) := \{(\pi, \theta_{out})\}$ such that $\{(\pi, \theta_{out})\} = \mathit{PSymExec}(p, \theta, \phi)$.*

For a given buggy program, a suspicious expression and an input condition, we infer the following specification:

Definition 14 (Specification)**.** *Let p be a program, e be a program expression, ϕ be an input condition, θ is a symbolic state over variables $\alpha_1, ..., \alpha_k$. A* specification *is a set of triples $\mathit{Spec}(p, e, \theta, \phi) := \{(\pi, \theta_c, \theta_{out})\}$ such that $\{(\pi, \theta_{out})\} = \mathit{PSymExec}(p', \theta, \phi)$, where $p' := p[e \mapsto \texttt{choose}()]$, $\texttt{choose}()$ is a function that returns a fresh symbolic variable β_i each time it is executed. For each path π, θ_c indicates a symbolic state in the context of which the function $\texttt{choose}()$ is called when the program is symbolically executed along π.*

We say that a summary $\mathit{Sum}(p, \theta, \phi)$ is *complete* if for each input σ satisfying the condition ϕ one of the following holds:

- $\mathit{Exec}(p, \sigma) = \omega$;
- $\exists(\pi, \theta_{out}) \in \mathit{Sum}(p, \theta, \phi).\ [\![\pi]\!]_\sigma = \mathit{True}$.

The completeness of a specification $\mathit{Spec}(p, e, \theta, \phi)$ can be defined in a similar manner.

In order to simplify further definitions, we assume (without loss of generality) that all formulas contain only a single variable α representing program inputs and a single variable β representing the values of the replaced program expression.

5.2.2 Verification condition

To check program equivalence, we construct a verification condition for a given patch using the inferred specification. Ideally, this condition should express the property "for each input satisfying a given input condition, if there is a path in the reference program followed by this input, then there should be a path in the patched program followed by this input and the outputs produced along these paths are equal". We refer to this condition as strict.

Definition 15 (Strict verification condition)**.** *Let p_r be a reference program, p_b be a buggy program, e be a suspicious expression in p_b, e' be a candidate patch, ϕ be an input condition, θ_{in} is a symbolic state over the variable α. A* strict verification condition *VC_{strict} for the program $p_b[e \mapsto e']$ is defined as follows:*

$$\forall\alpha\exists\beta \bigwedge_{(\pi^r,\theta^r_{out})}(\pi^r \Rightarrow \bigvee_{(\pi^b,\theta^b_c,\theta^b_{out})} \pi^b \wedge \beta = [\![e']\!]_{\theta^b_c} \wedge \theta^r_{out} = \theta^b_{out})$$

where the symbolic summary $\{(\pi^r, \theta^r_{out})\} \coloneqq \mathit{Sum}(p_r, \theta_{in}, \phi)$ and the specification $\{(\pi^b, \theta^b_c, \theta^b_{out})\} \coloneqq \mathit{Spec}(p_b, e, \theta_{in}, \phi)$.

However, the above condition cannot be used in many practical situations, where the existing symbolic execution engines reach their limits. For instance, we have to restrict the number of explored paths by performing loop unrolling in up to k iterations. As a result, the inferred specification is incomplete, and the introduced strict verification condition may classify two equivalent programs as non-equivalent. For example, if an input is captured by some path condition π^r, but not captured by any π^b, then the programs will be considered non-equivalent. To address this, we use a more practical version of the verification condition that we refer to as liberal.

Definition 16 (Liberal verification condition)**.** *Let p_r be a reference program, p_b be a buggy program, e be a suspicious expression in p_b, e' be a candidate patch, ϕ be an input condition, θ_{in} is a symbolic state over the variable α. A* liberal verification condition *$VC_{liberal}$ for the program $p_b[e \mapsto e']$ is defined as follows:*

$$\forall \alpha \exists \beta \bigwedge_{(\pi^r, \theta^r_{out})} \bigwedge_{(\pi^b, \theta^b_c, \theta^b_{out})} \pi^r \wedge \pi^b \wedge \beta = [\![e']\!]_{\theta^b_c} \Rightarrow \theta^r_{out} = \theta^b_{out}$$

where the symbolic summary $\{(\pi^r, \theta^r_{out})\} \coloneqq Sum(p_r, \phi, \theta_{in})$ and the specification $\{(\pi^b, \theta^b_c, \theta^b_{out})\} \coloneqq Spec(p_b, e, \phi, \theta_{in})$.

Compared with the strict verification condition, the liberal one only requires that for all intersections between a path condition π^r in the reference program and π^b in the buggy program (i.e. inputs satisfying $\pi^r \wedge \pi^b$), the symbolic outputs are the same in both programs. In the other words, this verification condition only checks equivalence of the functionality for which a specification has been inferred from both programs.

5.2.3 Patch generation

To implement a scalable patch generation that enforces conditional equivalence of the reference and the buggy programs, we propose a methodology of *counterexample-guided inductive repair* (CEGIR) that effectively combines counterexample-guided inductive synthesis (CEGIS) [3] and a patch synthesis algorithm of Angelix described in Chapter 4.

The overall workflow of the patch generator is shown in Figure 5.4 and illustrated by an example in Section 5.1. It performs a counterexample-guided refinement loop starting from the original expression as the initial candidate patch. The loop combines three modules: a conditional equivalence checker, an angelic forest extractor and a patch synthesizer that are described in details below.

Conditional equivalence checker In order to verify that a given candidate patch makes the buggy program conditionally equivalent to the reference program, we solve the liberal verification condition given in Definition 16. In order to solve the universally-quantified formula, we check the unsatisfiability of its negation, so as in previous works [51, 79]. Note that the introduced verification condition is an $\forall\exists$ formula, therefore its negation also produces a universally-quantified formula. However, the quantifiers $\forall\exists$ can be replaced with $\forall\forall$, since for each α the values of β is uniquely identified by the constraint $\pi^b \wedge \beta = [\![e']\!]_{\theta^b_c}$. Thus, the negation of the liberal verification condition in Definition 16 is

$$\exists\alpha\exists\beta \bigvee_{(\pi^r,\theta^r_{out})} \bigvee_{(\pi^b,\theta^b_c,\theta^b_{out})} \pi^r \wedge \pi^b \wedge \beta = [\![e']\!]_{\theta^b_c} \wedge \theta^r_{out} \neq \theta^b_{out}$$

The above formula is an $\exists\exists$ formula, therefore it can be solved using an off-the-shelf SMT solver. If the formula is unsatisfiable, then the patch is correct (conditionally equivalent to the reference program). Otherwise, a counterexample test is generated.

Angelic forest extractor Given a counterexample test and a specification inferred from the buggy program, our algorithm computes a compact specification for the expression based on angelic values (angelic forest). The algorithm of angelic forest extraction is similar to that used in Angelix [70]. It is presented in Algorithm 3. If angelic values cannot be extracted, then the bug cannot be fixed at the considered location (or the specification is incomplete). Otherwise, the values are extracted and passed to the synthesizer.

Patch synthesizer Since the input to the synthesizer is an angelic forest, we used the Angelix implementation of a patch synthesizer that extends SMT-based component-based program synthesis [70]. If a patch cannot be synthesized, then the search space (the set of considered transformations) is insufficient to find a repair. If a patch is found, it is passed to the conditional equivalence checker.

Proposition 1 (Correctness guarantee)**.** *Let p_b be a buggy program, p_r be* *a reference program, ϕ be an input condition, e be a suspicious expression in p_b. Assume that e' is a patch that is produced by the CEGIR algorithm (Figure 5.4) given complete specifications Sum(p_r,θ,ϕ) and Spec(p_b,e,θ,ϕ) as inputs. Then, $p_b[e \mapsto e']$ and p_r are conditionally partially equivalent w.r.t. the condition ϕ.*

5.3 Implementation

We have implemented the tool SemGraft for evaluating our technique. SemGraft consists of three main components: *preconditioned symbolic executor*, *verification condition generator* and *patch generator*. SemGraft receives a buggy and a reference program, an input condition and a test suite as input, and produces a patch for the buggy program as the output. Figure 5.5 shows the architecture of our tool. Below, we explain how these components are implemented.

Preconditioned symbolic executor (PSE) PSE is built on top of KLEE [12] — a widely used symbolic execution engine. To support preconditioned symbolic execution, the modified version of KLEE takes the user-defined input condition ϕ in SMTLIB2 format as input. Specifically, we modify the function *fork* of the KLEE interpreter which is called when KLEE encounters a symbolic branch. The path conditions of both branches are conjoined with the input condition to determine whether a path is terminated immediately or further explored. We integrate Z3 solver [21] with KLEE and pass symbolic constraints between them for checking the satisfiability of symbolic

Table 5.1: Busybox subject programs

Buggy Prog.	**Buggy Commit**	**Ref. Prog.**	**Ref. Prog. Version**	**Failure Description**	**Angelix'**	**SemGraft**
sed	c35545a	sed of GNU sed	version 3.01	Failed to handle zero-length match	Correct	Correct
seq	f7d1c59	seq of Coreutils	version 6.10	Wrong output when 2 input arguments are equal	Correct	Correct
sed	7666fa1	sed of GNU sed	version 3.01	Wrong output when handling `s///NUM`	Incorrect	Correct
sort	d1ed3e6	sort of Coreutils	version 8.27	Wrong output when handling `-kSTART,N.ENDCHAR`	Incorrect	Correct
seq	d86d20b	seq of Coreutils	version 8.27	`seq` no longer accepts 0 value as increment argument	Incorrect	Correct
sed	3a9365e	sed of GNU sed	version 3.01	Failed to handle `s///` which has empty matches	Incorrect	Correct

expressions. PSE outputs symbolic formula in SMTLIB2 format and invokes Z3 solver via a wrapper function.

Verification condition generator (VCG) VCG takes the symbolic summary of the reference program and the specification of the buggy program, both of which are obtained by executing PSE with the input condition. Our tool SemGraft supports both kinds of verification condition as per Subsection 5.2.2, but the default option of VCG is the liberal verification condition which is more practical for real-world programs. We use an open-source library jSMTLIB[1] for processing SMT files generated by PSE.

Patch generator (PG) PG takes the liberal verification condition in SMTLIB2 format and executes a counterexample-guided inductive repair loop until it finds a patch that satisfies the desired property. The workflow of PG is shown in Figure 5.4.

5.4 Evaluation

To evaluate the effectiveness of our approach, we aim to investigate the following research questions:

[1]jSMTLIB website: `https://github.com/smtlib/jSMTLIB`

Table 5.2: Coreutils subject programs

Buggy Prog.	Buggy Commit	Ref. Prog.	Ref. Prog. Version	Failure Description	Angelix'	SemGraft
mkdir	f7d1c59	mkdir of Busybox	version 1.27.2	Segmentation fault	Incorrect	Correct
mkfifo	cdb1682	mkfifo of Busybox	version 1.27.2	Segmentation fault	Incorrect	Correct
mknod	cdb1682	mknod of Busybox	version 1.27.2	Segmentation fault	Incorrect	Correct
copy	f3653f0	copy of Busybox	version 1.27.2	Failed to copy a file	Correct	Correct
md5sum	739cf4e	md5sum of Busybox	version 1.27.2	Segmentation fault	Correct	Correct
cut	6f374d7	cut of Busybox	version 1.27.2	Failed to handle `-b 2-,3-` like `-b 2-`	Incorrect	Correct

(RQ1) Can SemGraft generate repairs for real-world software?

(RQ2) Can SemGraft alleviate the overfitting problem of existing test suite based program repair techniques using the reference implementation?

RQ1 is designed to investigate the applicability of our approach for repairing real-world applications. A previous study [52] has reported that a straightforward combination of a state-of-the-art equivalence checking system with counterexample-guided inductive synthesis scales only to small programs. To be applicable to real-world programs such as Busybox and GNU Coreutils, our approach sacrifices discovers a partial specification for checking conditional equivalence w.r.t. a user-provided condition. Therefore, we also discuss how such a condition can be derived from existing tests.

RQ2 assesses the correctness of generated patches compared with test-driven program repair approaches. As in previous works [63, 70], we identify a generated patch as *correct* only if it is *syntactically* equivalent to the developer patch (modulo trivial refactorings).

5.4.1 Experimental setup

In order to address the described research questions, we choose the subjects in our experiments according to the following four criteria.

1. The subjects are real-world software that is widely used.
2. Reference programs are available that process the same inputs as the buggy programs but exhibit the correct behavior.
3. The buggy and the reference program are substantially different in their structure.
4. The developer patches are within the search spaces of our implementation. By the search space we mean the set of considered transformations defined through the components used for component-based synthesis.

The last criteria allows us to reason about correctness of generated patches (e.g. if the developer patch was not in the search space, then any generated patch would *a priori* be identified as incorrect).

Our subjects are 12 real software errors of two open-source C projects Busybox[2] and GNU Coreutils[3] extracted from (1) commit logs, (2) bug reports and (3) previous research [12]. Both Busybox and GNU Coreutils provide many common UNIX utilities, but Busybox has been implemented with size-optimization, limited resources, and is mainly used for small or embedded systems. We employ our tool SemGraft to repair the embedded Linux Busybox with GNU tools like Coreutils as reference, and vice versa.

To address the second question (RQ2), we compared our technique with a state-of-the-art test-driven program repair approach, Angelix [70]. Angelix is closely related to our technique since it also applies symbolic execution to infer specification and synthesizes patches. We selected this approach for our

[2]Busybox: `https://busybox.net/about.html`
[3]GNU Coreutils: `https://www.gnu.org/software/coreutils/`

evaluation because this enables us to more objectively investigate the impact of specification inferred from a reference program. Specifically, since our implementation reuses the synthesizer of Angelix, both Angelix and SemGraft explore the same space of candidate patches. In order to ensure that the systems have access to the same semantic information about the program, we integrated the algorithm of Angelix into SemGraft in such a way that both tools use the same specification inferred from the buggy program (we refer to this version of Angelix as Angelix'). The main difference of the two tools is that SemGraft performs a counterexample-guided inductive repair loop to ensure conditional equivalence with the reference implementation, while Angelix' relies solely on the test suite provided by GNU Coreutils/Busybox developers and stops immediately when finding an expression satisfying the tests.

All our experiments were performed on Intel Xeon CPU E5-2630 v4 @ 2.20GHz CPU with Ubuntu 14.04 64-bit operating system.

5.4.2 Summary of experiments

Table 5.1 summarizes our experiments with Busybox and Table 5.2 summarizes our experiments with GNU Coreutils. For each pair of a buggy and a reference program, the tables show the name of the buggy program and its version in the commit history, the reference program and its version, a description of the bug, and the results of executing Angelix' and SemGraft for repairing the defect.

SemGraft demonstrated that the proposed approach can be applied to real-world programs. Specifically, it managed to repair all defects that are repaired by Angelix. Since the workflow of SemGraft also includes inferring specification for a reference program and checking verification conditions, it required a longer time to generated patches. Specifically, Angelix' required 15 minutes on average to generate patches, while SemGraft required 45 minutes.

In the experiments, SemGraft inferred specifications consisting of up to 81 paths from a reference program and up to 250 paths in a buggy program. The number of paths, in general, depends on the structure of the buggy and the reference programs, bounds used for symbolic execution and the chosen condition ϕ. Typically, the specification inferred from the buggy program includes more paths due to additional symbolic variables injected into the buggy program for suspicious expressions.

As can be seen from the tables, SemGraft generated repairs equivalent to developer patches for all considered examples, while Angelix' that relies only on tests repaired less than half of the defects correctly. This shows that the reference implementation indeed can help to alleviate test overfitting.

5.4.3 Deriving input condition

An input condition used for enforcing conditional equivalence of the patched and the reference program is an important part of our approach and it has to be defined by the user. We use an example of a bug in `cut` (ver. 6f374d7) to demonstrate how such a condition can be defined in practice. `cut` extracts sections from each line of its input. The buggy version of `cut` of GNU Coreutils wrongly interprets the command `-b 2-,3-` as `-b 3-` (extract input bytes staring from the third byte) instead of `-b 2-` (extract input bytes staring from the second byte). The developer provides the following two tests that cover the buggy functionality:

```
echo -ne '1234' | cut -b 2-,3-
echo -ne '1234' | cut -b 3-,2-
```

For both of these tests, the expected output is `234`. The above two tests cover program behavior for two concrete pairs of indexes $(2, 3)$ and $(3, 2)$. However, this is insufficient for a test-driven program repair to produce a patch that *generalizes* beyond the tests.

In this approach, we propose to define an input condition for generating conditional equivalence-enforcing patches by *parametrizing* existing tests. Note that the purpose of parametrizing the test is to make generated patches generalize, yet ensuring tractability of specification inference. Therefore, the user should parametrize the essential part in the test related to the failure. In this case, we inject parameters instead of the indexes $\{2, 3\}$ that affect the way the data is processed. As a result, we obtain the following input:

```
echo -ne '1234' | cut -b α0-,α1-
```

where α_0 and α_1 indicate the injected parameters. Given such a parametrization, the condition ϕ will be accordingly defined as:

$$\begin{aligned}\phi :=& \mathit{arg0}[0] = \text{“}-\text{”} \wedge \mathit{arg0}[1] = \text{“}b\text{”} \wedge \\ & \mathit{arg1}[1] = \text{“}-\text{”} \wedge \mathit{arg1}[2] = \text{“},\text{”} \wedge \mathit{arg1}[4] = \text{“}-\text{”} \wedge \mathit{in} = \text{“1234”}\end{aligned}$$

where *arg0* and *arg1* denote command-line arguments, *in* is the standard input stream.

This example demonstrates that defining an input condition for our approach may require a small effort from the user. However, we believe that such a condition can be potentially derived automatically. One possible way to do that would be to execute the failing test *concolically* to collect input constraints (e.g. using ZESTI [64]), and construct an input condition for our approach by generalizing the obtained constraint. We leave this for future work.

5.4.4 Impact of reference program

In this section, we show how the use of a reference implementation can enable SemGraft to find a correct path, while Angelix' finds a plausible (passing the given tests), but incorrect repair.

For the discussed bug in `cut` program, Angelix' uses the tests given above to construct a patch in Figure 5.6a. This patch adds a condition into the program that changes the way how indexes in the given command are handled. The expression includes the disjunct `eol_range_start == 3` that ensures that the index 3 is not used by the command `-b 2-,3-`. However, this condition does not generalize to other values of the indexes that can appear in such command but merely overfit the given test.

As opposite to Angelix', SemGraft extracts a specification from the buggy and the reference programs via preconditioned symbolic execution with ϕ. In this example, it extracts 30 paths from the buggy program and 18 paths from the donor program (Busybox `cut`). Then, it performs a counterexample-guided inductive repair loop until it finds a patch that enforces conditional equivalence of Coreutils `cut` and Busybox `cut` w.r.t. ϕ. Specifically, after obtaining a candidate patch as in Figure 5.6a, it generates a counterexample test `-b 3-,4-` for which the output of Coreutils `cut` diverges from Busybox `cut`. Based on this test, it extracts the angelic path

$$\begin{aligned}\{\ &(\beta^0, \mathit{True}, \{\texttt{initial} \mapsto 3, \texttt{eol\_range\_start} \mapsto 0, \}),\\ &(\beta^1, \mathit{False}, \{\texttt{initial} \mapsto 4, \texttt{eol\_range\_start} \mapsto 3, \})\ \}.\end{aligned}$$

Given the extracted path, SemGraft generates the patch in Figure 5.6b, which is identical to the developer repair. SemGraft also proves that it is correct (equivalent to Busybox `cut`) for all possible combinations of indexes in the described command.

```
int search(int x, int a[], int length){
  int i;
  for (i=0; i<length; i++) {
    if (x == a[i])
      return i;
  }
  return -1;
}
```

(a) Reference program

ID	$\pi^{\mathbf{r}}$	$\theta^{\mathbf{r}}_{out}$
r_1	$\gamma = \alpha_0$	0
r_2	$\gamma \neq \alpha_0 \wedge \gamma = \alpha_1$	1
r_3	$\gamma \neq \alpha_0 \wedge \gamma \neq \alpha_1 \wedge \gamma = \alpha_2$	2
r_4	$\gamma \neq \alpha_0 \wedge \gamma \neq \alpha_1 \wedge \gamma \neq \alpha_2$	-1

(b) Summary of reference program

```
int search(int x, int a[], int length){
  int L = 0;
  int R = length-1;
  do {
    int m = (L+R)/2;
    if (x == a[m]) {
      return m;
    } else if (x < a[m]) { // x > a[m]
      L = m+1;
    } else {
      R = m-1;
    }
  } while (L <= R);
  return -1;
}
```

(c) Buggy program

ID	$\pi^{\mathbf{b}}$	$\theta_{\mathbf{c}}$	$\theta^{\mathbf{b}}_{out}$
b_1	$\gamma = \alpha_1$	-	1
b_2	$\gamma \neq \alpha_1 \wedge \beta^0 \wedge \gamma = \alpha_2$	$\beta^0 : \{x \mapsto \gamma, a[m] \mapsto \alpha_1\}$	2
b_3	$\gamma \neq \alpha_1 \wedge \beta^0 \wedge \gamma \neq \alpha_2 \wedge \beta^1$	$\beta^0 : \{x \mapsto \gamma, a[m] \mapsto \alpha_1\}$ $\beta^1 : \{x \mapsto \gamma, a[m] \mapsto \alpha_2\}$	-1
b_4	$\gamma \neq \alpha_1 \wedge \beta^0 \wedge \gamma \neq \alpha_2 \wedge \neg\beta^1$	$\beta^0 : \{x \mapsto \gamma, a[m] \mapsto \alpha_1\}$ $\beta^1 : \{x \mapsto \gamma, a[m] \mapsto \alpha_2\}$	-1
b_5	$\gamma \neq \alpha_1 \wedge \neg\beta^0 \wedge \gamma = \alpha_0$	$\beta^0 : \{x \mapsto \gamma, a[m] \mapsto \alpha_1\}$	0
b_6	$\gamma \neq \alpha_1 \wedge \neg\beta^0 \wedge \gamma \neq \alpha_0 \wedge \beta^1$	$\beta^0 : \{x \mapsto \gamma, a[m] \mapsto \alpha_1\}$ $\beta^1 : \{x \mapsto \gamma, a[m] \mapsto \alpha_0\}$	-1
b_7	$\gamma \neq \alpha_1 \wedge \neg\beta^0 \wedge \gamma \neq \alpha_0 \wedge \neg\beta^1$	$\beta^0 : \{x \mapsto \gamma, a[m] \mapsto \alpha_1\}$ $\beta^1 : \{x \mapsto \gamma, a[m] \mapsto \alpha_0\}$	-1

(d) Specification of buggy program

Negative input	$\{x \mapsto 2, a \mapsto [2,4,6], \mathit{length} \mapsto 3\}$
Expected output	0
Symbolic inputs	$\{x \mapsto \gamma, a \mapsto [\alpha_0, \alpha_1, \alpha_2], \mathit{length} \mapsto \delta\}$
Input condition	$\phi \coloneqq \alpha_0 < \alpha_1 < \alpha_2 \wedge \delta = 3$

(e) Test and input condition

Figure 5.2: SemGraft motivating example.

$$VC = \forall\alpha_0\forall\alpha_1\forall\alpha_2\forall\gamma \bigwedge_{(\pi^r,\theta^r_{out})} \bigwedge_{(\pi^b,\theta^b_{out})} \pi^r \wedge \pi^b \wedge (\beta = e[\![\theta_c]\!]) \Rightarrow \theta^r_{out} = \theta^b_{out}$$

$$\begin{aligned}
&\equiv \forall\alpha_0\forall\alpha_1\forall\alpha_2\forall\gamma(\\
{}^{(r_1,b_3)}\quad & \neg(\gamma = \alpha_0 \wedge \gamma \neq \alpha_1 \wedge \beta^0 \wedge \beta^0 = \gamma < \alpha_1 \wedge \gamma \neq \alpha_2 \wedge \beta^1 \wedge \beta^1 = \gamma < \alpha_2) \\
{}^{(r_1,b_4)}\quad & \wedge \neg(\gamma = \alpha_0 \wedge \gamma \neq \alpha_1 \wedge \beta^0 \wedge \beta^0 = \gamma < \alpha_1 \wedge \gamma \neq \alpha_2 \wedge \neg\beta^1 \wedge \beta^1 = \gamma < \alpha_2) \\
{}^{(r_3,b_6)}\quad & \wedge \neg(\gamma = \alpha_2 \wedge \gamma \neq \alpha_1 \wedge \neg\beta^0 \wedge \beta^0 = \gamma < \alpha_1 \wedge \gamma \neq \alpha_0 \wedge \beta^1 \wedge \beta^1 = \gamma < \alpha_0) \\
{}^{(r_3,b_7)}\quad & \wedge \neg(\gamma = \alpha_2 \wedge \gamma \neq \alpha_1 \wedge \neg\beta^0 \wedge \beta^0 = \gamma < \alpha_1 \wedge \gamma \neq \alpha_0 \wedge \neg\beta^1 \wedge \beta^1 = \gamma < \alpha_0))
\end{aligned}$$

Figure 5.3: Verification condition

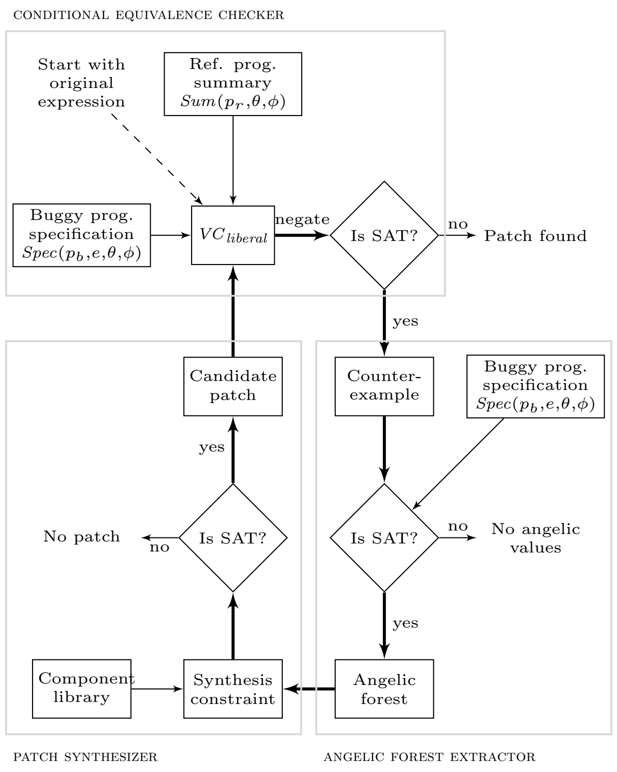


Figure 5.4: Counterexample-guided inductive repair.

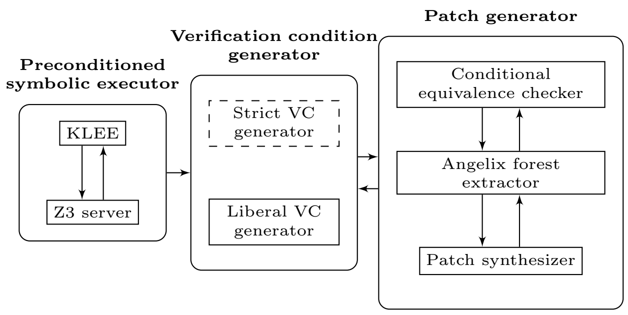


Figure 5.5: Architecture of SemGraft.

```
if (!rhs_specified)
  {
    if (eol_range_start == 0 || eol_range_start == 3)
      eol_range_start = initial;
    field_found = true;
  }
```

(a) Patch generated by Angelix' based on tests.

```
if (!rhs_specified)
  {
    if (eol_range_start == 0 || initial < eol_range_start)
      eol_range_start = initial;
    field_found = true;
  }
```

(b) Patch generated by SemGraft based on reference program.

Figure 5.6: Generated patches for `cut` (ver. 6f374d7).

Chapter 6

Symbolic execution with existential second-order constraints

Program repair algorithms presented in previous chapters rely on symbolic execution. One of the key limitations of symbolic execution is the path explosion problem, since in programs with loops symbolic execution might have to explore an infinite number of paths. To address this problem in the context of program repair, we describe an extension of symbolic execution that can take the space of patches into account to prune irrelevant paths.

In symbolic execution, program inputs are assigned symbolic variables instead of concrete values. The result of executing a program with symbolic inputs is a set of constraints over these symbolic variables called path conditions. Path condition of a program path captures all inputs that would drive the execution along this program path. Symbolic variables used in existing symbolic execution systems typically range over numbers, arrays and strings.

We introduce symbolic execution with existential second-order constraints

(SE-ESOC), that extends traditional symbolic execution by allowing symbolic variables to range over functions. A function in a program can be marked as "symbolic" via a second-order symbolic variable. Then, the goal of SE-ESOC is to *synthesize* an interpretation of this function that satisfies certain reachability properties of the analyzed program (the properties depend on the application). SE-ESOC collects constraints on second-order variables and solves them through program synthesis.

Example 1. *Assume that* `search(data, pred)` *returns the index of an element of the array* `data` *that satisfies the predicate* `pred`*. Consider the question "What predicate would make* `search` *return 2 given the array* `[0, 1, 2]`*?". SE-ESOC can answer this question by executing* `search([0, 1, 2],` ρ`)` *symbolically with a second-order variable* ρ*, and synthesizing e.g.* $\rho \coloneqq \lambda x.\ x > 1$*.*

Contrary to works [33] utilizing the theory of uninterpreted functions, SE-ESOC aims to discover *implementations* of symbolic functions. Thus, SE-ESOC takes in a *language of interpretations* for second-order variables. Similar to the syntax-guided program synthesis approach [3], a language of interpretations is defined in our approach via a context-free grammar, and a size bound.

In the context of program repair, suspicious statements in the buggy program can be replaced with second-order symbolic variables. Thus, a fragment of code in a program can be abstracted as a second-order symbolic variable. Instantiations of the second-order variable then amount to alternate code fragments to replace the current one, thereby bringing out the connection

```
size_t search(int data[],
              size_t len,
              int (*pred)(int)) {
  size_t i;
  for (i = 0; i < len; i++)
    if (pred(data[i]))
      return i;
  return len;
}
```

(a) Search function.

Executing `search` with symbolic second-order input ρ:

```
search((int[]){0, 1, 2}, 3, ρ);
```

Symbolic input state:

$$\theta_{in} := \{\mathtt{data}[0] \mapsto 0, \mathtt{data}[1] \mapsto 1, \mathtt{data}[2] \mapsto 2, \mathtt{len} \mapsto 3, \mathtt{pred} \mapsto \rho\}$$

Symbolic execution results:

Path condition π	Generated input	Output state θ_{out}
$\rho(0)$	$\{\rho \mapsto \lambda x.\ \mathit{true}\}$	$\{\mathtt{return} \mapsto 0, ...\}$
$\neg\rho(0) \wedge \rho(1)$	$\{\rho \mapsto \lambda x.\ x > 0\}$	$\{\mathtt{return} \mapsto 1, ...\}$
$\neg\rho(0) \wedge \neg\rho(1) \wedge \rho(2)$	$\{\rho \mapsto \lambda x.\ x > 1\}$	$\{\mathtt{return} \mapsto 2, ...\}$
$\neg\rho(0) \wedge \neg\rho(1) \wedge \neg\rho(2)$	$\{\rho \mapsto \lambda x.\ \mathit{false}\}$	$\{\mathtt{return} \mapsto 3, ...\}$

(b) SE-ESOC.

Figure 6.1: Testing search function via traditional SE and SE-ESOC.

between SE-ESOC and program repair. SE-ESOC can directly synthesize a patch by finding interpretations of the symbolic functions.

The proposed technique alleviates the path explosion problem in program repair algorithms relying on first-order symbolic execution. Program repair techniques such as SemFix [73] and Angelix [70] split patch generation into two steps. First, they replace suspicious statements with first-order symbolic variables and infer specification via symbolic execution. As a second step, they synthesize patches that satisfy the inferred specification. An important limitation of these approaches is that they have to potentially explore an infinite number of paths e.g. if the suspicious statements are inside a loop. However, by raising the order of path constraints, we can efficiently prune irrelevant paths. The pruning is achieved by avoiding paths that are infeasible in the context of considered language of interpretations (the space of patches).

To implement SE-ESOC, it is sufficient, in principle, to apply a syntax-guided synthesizer [3] for solving queries with second-order variables. However, existing synthesis algorithms are not suitable for this application. SMT solvers used in symbolic execution engines [12, 103] cannot solve the considered kind of second-order constraints, however our goal was to support second-order variables without switching to a specialized solver. The reason for not switching to specialized solvers is that we might have second-order variables as well as first-order variables in various theories in a single path condition. Then, a suitable approach to support second-order variables is to encode second-order formulas through first-order formulas as proposed by Jha et al. [38]. However, our experiments demonstrated that the mentioned encoding provides highly inefficient unsatisfiability proofs. Meanwhile, the performance of symbolic execution critically depends on the performance of unsatisfiable queries for on-the-fly pruning of infeasible paths. To address the above challenges, we introduce a new method of second-order constraint solving that relies on propositional encoding, which substantially improves the efficiency of unsatisfiability proofs compared with previous techniques.

6.1 Overview

This section describes (1) second-order formulas considered in this thesis (2) the difference between traditional SE and SE-ESOC and (3) an application of SE-ESOC to program repair.

6.1.1 Second-order formulas

We view second-order constraint solving as an instance of program synthesis [20]. Formally, we consider second-order formulas with existentially quantified second-order variables, and a Henkin (non-standard) semantics [71] of satisfiability (Definition 17). Specifically, each second-order variable is associated with a domain of interpretations defined via a user-provided language.

Example 2. *Assume that ρ is a second-order variable whose domain is restricted by the language LIA defined as follows:*

$\langle \mathit{Term} \rangle$ ::= $\langle \mathit{Var} \rangle$ | $\langle \mathit{Constant} \rangle$

| $\langle \mathit{Term} \rangle$ '+' $\langle \mathit{Term} \rangle$ | $\langle \mathit{Term} \rangle$ '-' $\langle \mathit{Term} \rangle$ | $\langle \mathit{Constant} \rangle$ '*' $\langle \mathit{Term} \rangle$

Then, $\rho(0) > 0 \wedge \rho(1) \leq 0$ is satisfiable by $\rho := \lambda x.\ 1 - x$, while $\rho(0) > 0 \wedge \rho(1) \leq 0 \wedge \rho(2) > 0$ is unsatisfiable since all functions in LIA are monotonic.

Since we rely on a non-standard semantics of satisfiability, we cannot use the theory of uninterpreted functions supported by most SMT solvers as in previous works [33]. Thus, we have to add support for this semantics in an existing SMT solver. To make the problem more tractable, we bound the size of interpretations by a user-defined constant D. However, even with this restriction, an integration of second-order solving with symbolic execution remains challenging, since existing synthesis algorithms are not optimized for unsatisfiable queries [20]. This motivated us to design a new SMT-based synthesis method described in Section 6.2.1.

6.1.2 Comparing SE-ESOC with traditional SE

Consider the function `search` in Figure 6.1a. This function takes an array `data`, a value `len` representing its length, a pointer to a predicate function `pred`, and returns the index of the first element of the array that satisfies the predicate.

In traditional symbolic execution, numeric inputs are replaced with logical variable as shown for the elements $\alpha_1, \alpha_2, \alpha_3$ of the array in Figure 2.1b. Assume that the predicate `pred` is a function `pos` that checks if a given value is positive. In this context, symbolic execution explores four paths as shown in the table in Figure 2.1b, in which the path conditions are constraints over the variables $\alpha_1, \alpha_2, \alpha_3$. The corresponding test inputs are concrete values of the elements of the array: $\{1, 0, 0\}$, $\{0, 1, 0\}$, $\{0, 0, 1\}$ and $\{0, 0, 0\}$.

In contrast to traditional symbolic execution, SE-ESOC enables us to explore possible program executions depending on the definition of the predicate `pred`. Assume that `pred` is represented by a variable ρ, for which the language of interpretations is as follows:

$\langle \mathit{BoolTerm} \rangle$ ::= $\langle \mathit{Term} \rangle$ '`>`' $\langle \mathit{Term} \rangle$ | $\langle \mathit{Term} \rangle$ '`>=`' $\langle \mathit{Term} \rangle$
  | $\langle \mathit{Term} \rangle$ '`=`' $\langle \mathit{Term} \rangle$ | '`true`' | '`false`'

where `Term` is defined in Example 2. Then, the path conditions are constraints over ρ as shown in the table in Figure 6.1b. The corresponding test

```
scanf("%d",&x);
for (i=0;i<10;i++) {
  int t = x - i;
  if (t>0)
    printf("1");
  else
    printf("0");
}
```

(a) Buggy program.

```
scanf("%d",&x);
for (i=0;i<10;i++) {
  int t = α;
  if (t>0)
    printf("1");
  else
    printf("0");
}
```

(b) Symbolic state.

```
scanf("%d",&x);
for (i=0;i<10;i++) {
  int t = ρ(i,x);
  if (t>0)
    printf("1");
  else
    printf("0");
}
```

(c) Symb. function.

$$\pi_1 \coloneqq \alpha_1 > 0 \land \alpha_2 > 0 \land ...$$
$$\pi_2 \coloneqq \alpha_1 \leq 0 \land \alpha_2 > 0 \land ...$$
$$\pi_3 \coloneqq \alpha_1 > 0 \land \alpha_2 \leq 0 \land ...$$

(d) First-order PCs.

$$\pi_1 \coloneqq \rho(0,5) > 0 \land \rho(1,5) > 0 \land ...$$
$$\pi_2 \coloneqq \rho(0,5) \leq 0 \land \rho(1,5) > 0 \land ...$$
$$\pi_3 \coloneqq \rho(0,5) \leq 0 \land \rho(1,5) \leq 0 \land ...$$

(e) Second-order PCs.

Figure 6.2: Repairing program using different approaches.

inputs are interpretations of ρ: $\lambda x.\ \mathit{true}$, $\lambda x.\ x > 0$, $\lambda x.\ x > 1$ and $\lambda x.\ \mathit{false}$.

Note that it is possible to combine first-order and second-order symbolic variables in the same symbolic execution session by executing

$$\texttt{search}((\texttt{int}[])\{\alpha_1, \alpha_2, \alpha_3\}, 3, \rho)$$

Then, the synthesized predicates ρ will be parameterized by the variables $\alpha_1, \alpha_2, \alpha_3$.

6.1.3 Application to program repair

The goal of program repair is to modify a buggy program to eliminate the observable failures. Its important subtask is to fill a hole in the program (e.g. replace a buggy statement) to enable the program to satisfy the requirements (e.g. to pass the tests). We review existing approaches to solve this subtask relying on traditional SE, and show how SE-ESOC addresses their limitations.

Consider a program P in Figure 6.2a that reads a number, performs 10 loops iterations and, at each iteration, prints "0" or "1" depending on the sign of the variable `t`. For instance, for the input "5", it prints "1111100000". Assume that the correct output should be "1111111000", and our goal is to repair the program by replacing `x - i` with an expression from LIA (defined in Example 2) that would enable the program to pass the test (e.g. `x - i + 2`).

Semantics-based repair approaches [73, 70] infer a specification using symbolic execution, and synthesize a patch based on this specification. First, they replace the identified buggy expression with a symbolic variable α as shown in Figure 6.2b. Then, they symbolically execute the program with the input "5" and infer path conditions $\pi_1, \pi_2, ..., \pi_{1024}$ shown in Figure 6.2d. Finally, a patch is synthesized by solving the following second-order formula:

$$\exists e \in \mathit{Term}.\ (\bigvee_i \pi_i[\alpha \mapsto e]) \wedge \mathit{stdout} = \text{"1111111000"}$$

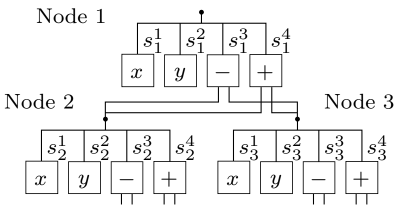
where *stdout* is a variable that captures the standard output of the application, $\pi_i[\alpha \mapsto e]$ is a formula obtained from π_i by substituting α with the term e. Such techniques suffer from the path explosion problem. For instance, there are 10 loop iterations and therefore the algorithm has to explore 1024 paths, as shown in Figure 6.2d.

We now demonstrate how SE-ESOC can be used to address the aforementioned limitation of previous techniques. Instead of using first-order variables α to infer synthesis specification, we replace the buggy statement with a symbolic function ρ as shown in Figure 6.2c. Then, SE-ESOC is applied to directly synthesize a patch by finding an interpretation of ρ that satisfies a test-passing path. The key benefit of this approach is that it substantially reduces the number of explored paths. For the described example, it will explore at most 20 execution paths as shown in Figure 6.2e, and the rest of the paths are infeasible, which can be non-constructively proven as follows.

Proof. Among 1024 possible paths, there are 1004 paths with path conditions containing the clauses $\rho(l, 5) > 0, \rho(n, 5) \leq 0, \rho(m, 5) > 0$ or the clauses $\rho(l, 5) \leq 0, \rho(n, 5) > 0, \rho(m, 5) \leq 0$ for some $l < n < m$. All these path conditions are unsatisfiable since for any $\rho \in \mathit{Term}$, $\lambda x.\rho(x, 5)$ is monotonic. $\square$

Note that monotonic functions in this example are given for clarity. Our approach does not rely on monotonicity, and is effective in more general cases as shown experimentally in Section 6.3.2.

SE-ESOC enables a reduction of the number of explored paths since it takes the language of symbolic function interpretations into account, i.e. it is *syntax-guided*. The reduction of the number of explored paths has important implications, since it might increase the efficiency of program repair or increase the probability of finding a patch when repairing programs with loops.



Node 1
Node 2
Node 3
s_1^1 s_1^2 s_1^3 s_1^4
x y $-$ $+$
s_2^1 s_2^2 s_2^3 s_2^4
s_3^1 s_3^2 s_3^3 s_3^4

(a) Tree with "abstract" nodes.

$$s_1^1 \mapsto x$$
$$s_1^3 \wedge s_2^1 \wedge s_3^2 \mapsto x - y$$
$$s_1^4 \wedge s_2^1 \mapsto \{x + T\}_{T \in Term}$$

(b) Selectors to terms.

Figure 6.3: Encoding via propositional selector variables.

6.2 Methodology

In this section, we first formally describe second-order constraints used in our approach and a method of solving these constraints. Secondly, we demonstrate how second-order solving is integrated with symbolic execution, and describe implemented constraint optimizations. Finally, we show how the resulting technique can be applied for program repair.

6.2.1 Second-order solving

As is usual in SMT literature [9], we consider formulas and terms built from predicate and function symbols (e.g. "+", "−", " > ") from a given signature Σ. We denote the set of all such formulas and terms as L_Σ. We also consider a background theory $\mathcal{T}$ that fixes the interpretations of the symbols in Σ. In this work, we are interested in an extended set of formulas and terms $L_{\Sigma \cup \mathrm{P}}$ constructed from the symbols in Σ and an additional set of predicate and function symbols $\mathrm{P} \coloneqq \{\rho_1, ..., \rho_n\}$ without interpretations in $\mathcal{T}$, that we refer to as *second-order variables* (or symbolic functions).

For a formula ϕ over a second-order variable ρ and a term $t \in L_\Sigma$ with a designated set of variables $x_1, ..., x_n$, we say that a first-order formula $\phi[\rho \mapsto t]$ is a *substitution* of ρ with t, if it is obtained by replacing each sub-term $\rho(t_1, ..., t_n)$ of ϕ (for some terms t_1, ..., t_n) with a term computed as the result of beta-reduction of the lambda expression $(\lambda x_1...x_n.\ t)\ t_1\ ...\ t_n$. For instance, let ϕ be $\rho(a, 1) > 0$ and t be $x_1 + x_2$, then $\phi[\rho \mapsto t]$ is defined as $a + 1 > 0$.

Definition 17 (Second-order satisfiability)**.** *Let $\phi \in L_{\Sigma \cup \mathrm{P}}$ be a second-order formula, $\mathcal{L} : \mathrm{P} \to 2^{L_\Sigma}$ — a mapping from second-order variables to sub-languages of L_Σ — be domains of interpretations. Then, ϕ is satisfiable iff, for some terms $t_1 \in \mathcal{L}(\rho_1), ..., t_n \in \mathcal{L}(\rho_n)$, the first-order formula $\phi[\rho_1 \mapsto t_1, ..., \rho_n \mapsto t_n]$ is satisfiable w.r.t. $\mathcal{T}$.*

The key part of this definition is the domains of interpretations $\mathcal{L}$ that are sub-languages of L_Σ for each second-order variable. In our approach, the sub-languages are either provided by the user or by a tool/algorithm that relies on SE-ESOC. Particularly, a sub-language is defined as a pair (G, D) of a context-free grammar G with the symbols from Σ as terminals (as in SyGuS format [3]) and an integer value D that describes the maximum depth of considered terms (i.e. the maximum number of nodes in a path from the root of a term to its leaf).

Similar to the prior approach described in Section 2.4.1, we rely on a library of components to encode a space of terms from a given language of interpretations. Note that for a synthesis problem with a language defined via a pair (G, D), it is straightforward to encode it as a component-based synthesis problem by considering each grammar rule $N \to F(N_1, ..., N_n)$ (for non-terminals $N, N_1, ..., N_n$) of G as a component F with inputs $N_1, ..., N_n$. Thus, without the loss of generality, we assume later that, instead of a grammar G, our language is defined through a set of components $F_1, ..., F_C$.

One way to implement a solver for the considered kind of second-order formulas is to encode them through first-order formulas using e.g. the approach described in Section 2.4.1. However, this approach relies on linear integer arithmetic to encode a space of terms, which results in inefficient proofs of unsatisfiability. On the other side, SE-ESOC critically depends on the performance of unsatisfiable queries to avoid infeasible paths, as shown in Section 6.1.3.

In order to optimize unsatisfiable queries, we introduce an new encoding of second-order formulas through propositional *selector variables* instead of integer location variables. Intuitively, this increases the effectiveness of conflict clause learning in CDCL-based [93] SMT solvers [21, 17] and therefore significantly improves the performance on unsatisfiable queries, which is shown experimentally in Section 6.3.

The key idea of the introduced *propositional synthesis encoding* is to represent the space of terms constructed from a given library of components via a tree with "abstract" nodes as shown in Figure 6.3a. Specifically, each intermediate node of the tree has as many subnodes as the maximal number of inputs of a component in a given component library. Each leaf of the tree corresponds to components that have no inputs. The semantics of each node is defined through the semantics of a component activated via selector variables.

Assume that s_i^j is the j-th selector of the i-th node, $\texttt{out}_i$ is the output of i-th node, C is the number of components, F_j is the semantics if the j-th component (e.g. $\lambda xy.\ x + y$ for addition). For each node i with subnodes $i_1, i_2, ..., i_k$, a set of terms is encoded as $\psi_i := \psi_{node} \wedge \psi_{choice}$, such that

$$\psi_{node} := \bigwedge_{j \in [1,C]} s_i^j \Rightarrow \mathtt{out}_i = F_j(\mathtt{out}_{i_1}, \mathtt{out}_{i_2}, ..., \mathtt{out}_{i_k})$$

$$\psi_{choice} := \mathit{exactlyOne}(s_i^1, s_i^2, ..., s_i^C)$$

In this encoding, ψ_{node} describes how the output value of a node depends on the values of its subnodes, ψ_{choice} ensures that exactly one of the components is selected inside each node (the cardinality constraint *exactlyOne* is implemented using sorting networks [1]). A term is constructed from an assignment of selector variables using a function `Sval2Term` that at each node picks a component that is activated by the corresponding selector variable as shown in Figure 6.3b. For instance, the term $x - y$ is constructed by enabling the component $-$ of the node 1 (via the selector s_1^3), the component x of the node 2 (via the selector s_2^1), and the component y of the node 2 (via the selector s_3^2).

Using the above encoding, a second-order constraint solver can be implemented on top of a first-order solver. Specifically, for a given formula ϕ over a second-order variable ρ, we define the procedure `Encode` as follows:

- each occurrence of a subterm $\rho(t_1, ..., t_n)$ in ϕ (for some terms $t_1, ...t_n$) is assigned a unique index i;
- for each occurrence of a subterm $\rho(t_1, ..., t_n)$ in ϕ with index i (for some terms $t_1, ...t_n$), the formula ϕ is conjoined with $\psi_1^i \wedge ... \wedge \psi_m^i$, where

m is the number of tree nodes and the terms $t_1, ..., t_n$ are treated as components without inputs;

- each occurrence of a subterm $\rho(t_1, ..., t_n)$ in ϕ with index i (for some terms $t_1, ...t_n$) is replaced with the variable $\texttt{out}_1^i$ representing the root of the i-th tree in the encoding.

Using this procedure, a second-order formula is transformed into a first-order formula over selector variables, which can be solved using an off-the-shelf SMT solver. From any satisfying assignment of the selector variables, an interpretation of ρ that satisfies ϕ can be reconstructed using `Sval2Term`, as stated formally below:

Proposition 2. *For any assignment of selector variables $S := \{s_1 \mapsto b_1, ..., s_n \mapsto b_n\}$ that satisfies $\phi' :=$ `Encode`(ϕ), the assignment $\{\rho \mapsto$ `Sval2Term`$(S)\}$ satisfies ϕ.*

6.2.2 Extension of symbolic execution

Compared with traditional SE described in Algorithm 1, SE-ESOC modifies the function `isSatisfiable`. Specifically, it adds support for second-order constraints by implementing the approach described in Section 6.2.1. The function `isSatisfiable` encodes each query ϕ using `Encode` before passing it to the underlying SMT solver. Later, a model of ϕ can be reconstructed from the model computed by the underlying SMT solver using `Sval2Term`.

Definition 18 (Second-order infeasible paths)**.** *Let P be a program taking a function as an input, ρ be a second-order variable, and L be a sub-language* *of L_Σ. Then, a path along which SE-ESOC computes a path condition π by executing P with the symbolic input ρ is* infeasible *iff the second-order formula π is unsatisfiable w.r.t. the domain of interpretations $\{\rho \mapsto L\}$.*

This definition of infeasible path depends on the syntax of the language of interpretations L. This property is crucial for mitigating the path explosion as will be shown in Section 6.3.2.

6.2.3 Program repair

SE-ESOC can be used, among others, to synthesize patches for program defects or models for unavailable libraries. Similarly to prior works [73, 70], the workflow of both these applications consists of three steps: injecting second-order symbolic variables, performing specification inference, and synthesizing patches.

Symbolic variable injection. In the context of program repair, suspicious program statements are substituted with applications of symbolic functions to local program variables. Suspicious program statements can be identified using, for instance, statistical fault localization [42]. For each of the identified suspicious statements, we iteratively apply the following transformation schemas parameterized with second-order variable ρ:

- changing the right-hand side of an assignment:

$$\mathtt{x} := \mathtt{E}; \quad \mapsto \quad \mathtt{x} := \rho(\mathtt{v_1}, ..., \mathtt{v_n});$$

- changing a condition:

$$\texttt{if (E) \{...\}} \quad \mapsto \quad \texttt{if } (\rho(\texttt{v}_1, ..., \texttt{v}_\texttt{n}))\ \texttt{\{...\}}$$

- adding an if-guard:

$$\texttt{S;} \quad \mapsto \quad \texttt{if } (\rho(\texttt{v}_1, ..., \texttt{v}_\texttt{n}))\ \texttt{S;}$$

where `S` is a statement, `E` is an expression, and $\texttt{v}_1, ...\texttt{v}_\texttt{n}$ are visible program variables. Specifically, we adopted a recently proposed heuristics [112] to select up to 10 local program variables whose definitions are the closest to the considered suspicious location.

Specification inference. The purpose of specification inference is to collect constraints over the injected second-order variables such that any interpretation that satisfies these constraint would meet our requirements. Specifically, our goal is to find interpretations of the symbolic functions that would enable the program to pass given tests. Assume that P is the original program, and P' is a program obtained by injecting a second-order variable ρ into P. Assume also that $\{in_i, \phi_i\}_{i\in[0,n]}$ is a set of tests, where in_i is the input of the i-th test and ϕ_i is the test assertion. For each test i, we execute the program P' using SE-ESOC with the concrete input in_i and obtain a set of path conditions $\pi_1^i, ..., \pi_k^i$, which are constraints over the variable ρ. Then, the specification is defined as follows:

$$\bigwedge_{i=0}^{n}(\bigvee_{j=0}^{k}\pi_j^i) \wedge \phi_i \tag{6.1}$$

The above second-order formula captures the property that for each test with index i there should be at least one path π_j^i along which the test assertion ϕ_i holds.

Patch/model synthesis. To synthesize interpretations of symbolic functions that satisfies the formula (6.1), we apply the second-order constraint solving method described in Section 6.2.1. For program repair, these interpretations constitute patches.

Tests are typically insufficient to guarantee the correctness of patches, which causes the test-overfitting problem [95]. The proposed approach is orthogonal to the problem of test-overfitting, however it is straightforward to integrate it with existing techniques for alleviating overfitting such as synthesizing minimal change via maximum satisfiability [69] or applying anti-pattens [101] or applying correctness assertions — by conjoining the encoding with additional constraints over selector variables.

6.2.4 Implementation

We implemented SE-ESOC inside KLEE [12], a widely used symbolic execution engine for C programs. Firstly, we extended KLEE to support second-order variables and implemented the generation of second-order path conditions in the symbolic execution runtime. Secondly, we implemented the encodings described in Section 2.4.1 and Section 6.2.1 on top of the underlying SMT solver.

KLEE provides an intrinsic function `klee_make_symbolic` for injecting symbolic variables. For example, the following call marks the memory corresponding to the variable `foo` as symbolic.

```
klee_make_symbolic(&foo, sizeof(foo), "foo");
```

To let users introduce second-order variables, we added an intrinsic function `klee_apply_symbolic`. This function applies a symbolic function to program expressions. For instance, the following code can be used to inject a call of ρ in Example 6.2c:

```
int t = klee_apply_symbolic("rho", 2, (int[]){i, x});
```

where `"rho"` is the name of the second-order variable, `2` is the number of arguments, and `(int[]){i, x}` is the array of arguments.

6.3 Evaluation

This evaluation addresses the following research questions:

(RQ1) Does SE-ESOC reduce the number of explored paths compared with program repair techniques relying in first-order symbolic execution? Does it improve the effectiveness of program repair?

(RQ2) Does the introduced second-order solving method based on propositional encoding improve SE-ESOC performance compared to previous encodings?

Table 6.1: Subjects of DBGBench dataset

Program	Description	Defects
find	Search for files in directory hierarchy	14
grep	Search for lines containing match to specified pattern	13

6.3.1 Experimental setup

To address the research questions, we conducted experiments with programs from GNU Coreutils[1], mature and widely-used implementations of UNIX utilities included in the majority of Linux distributions, that have been also employed in previous symbolic execution studies [12, 76].

We used a recently introduced DBGBench dataset [11]. DBGBench is a collection of 27 bugs from GNU Findutils and GNU Grep shown in Table 6.1. We chose this benchmark because it contains real error in widely used software, and because this dataset was designed for evaluating, among others, program repair techniques.

To examine the effect of second-order constraints on the path explosion, we compared our approach with Angelix [70], a state-of-the-art program repair system that relies on first-order symbolic execution. Specifically, we used the following three configurations:

FO Angelix that relies on first-order symbolic execution.

SO/CBS SE-ESOC that uses the encoding by Jha et al. [38] for second-order constraint solving.

SO/PSE SE-ESOC that uses the introduced propositional encoding for second-order constraint solving.

[1]GNU Coreutils: `https://www.gnu.org/software/coreutils/`

$$
\begin{aligned}
\langle \mathit{Bool} \rangle &::= \langle \mathit{Int} \rangle \text{ `}\texttt{<}\text{' } \langle \mathit{Int} \rangle \mid \langle \mathit{Int} \rangle \text{ `}\texttt{<=}\text{' } \langle \mathit{Int} \rangle \mid \langle \mathit{Int} \rangle \text{ `}\texttt{==}\text{' } \langle \mathit{Int} \rangle \\
&\mid \langle \mathit{Bool} \rangle \text{ `}\texttt{||}\text{' } \langle \mathit{Bool} \rangle \mid \langle \mathit{Bool} \rangle \text{ `}\texttt{\&\&}\text{' } \langle \mathit{Bool} \rangle \mid \text{`}\texttt{!}\text{' } \langle \mathit{Bool} \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle \mathit{Int} \rangle &::= \langle \mathit{Var} \rangle \mid \langle \mathit{Constant} \rangle \\
&\mid \langle \mathit{Int} \rangle \text{ `}\texttt{+}\text{' } \langle \mathit{Int} \rangle \mid \langle \mathit{Int} \rangle \text{ `}\texttt{-}\text{' } \langle \mathit{Int} \rangle \mid \text{`}\texttt{-}\text{' } \langle \mathit{Int} \rangle
\end{aligned}
$$

(a) Boolean functions.

$$
\begin{aligned}
\langle \mathit{Term} \rangle &::= \langle \mathit{Var} \rangle \mid \langle \mathit{Constant} \rangle \\
&\mid \langle \mathit{Term} \rangle \text{ `}\texttt{+}\text{' } \langle \mathit{Term} \rangle \mid \langle \mathit{Term} \rangle \text{ `}\texttt{-}\text{' } \langle \mathit{Term} \rangle \mid \text{`}\texttt{-}\text{' } \langle \mathit{Term} \rangle \\
&\mid \langle \mathit{Bool} \rangle \text{ `}\texttt{?}\text{' } \langle \mathit{Term} \rangle \text{ `}\texttt{:}\text{' } \langle \mathit{Term} \rangle
\end{aligned}
$$

(b) Integer functions.

Figure 6.4: Language of interpretations (search space).

Table 6.2: SE-ESOC results.

Subject	Patch			Paths			Time			SAT/UNSAT	
	FO	SO/CBS	SO/PSE	FO	SO/CBS	SO/PSE	FO	SO/CBS	SO/PSE	SO/CBS	SO/PSE
find.091557f6	-	-	Correct	400	0	16	2m 35s	TO	2m 29s	3.5s/TO	2.2s/3.8s
find.24bf33c0	Plausible	Plausible	Plausible	2	2	2	2s	3s	3s	1.3/-	0.5s/-
find.24e2271e	Correct	Correct	Correct	24	24	24	39s	1m 1s	1m 43s	2.1s/-	2.2s/-
find.07b941b1	Plausible	Plausible	Plausible	11	11	11	29s	41s	31s	0.7s/-	0.5s/-
find.e6680237	Correct	Correct	Correct	46	46	46	56s	2m 41s	2m 23s	1.2s/-	1.1s/-
find.dbcb10e9	-	-	Correct	400	0	14	3m 31s	TO	3m 1s	3.0s/TO	1.7s/5.8s
find.e1d0a991	Plausible	Plausible	Plausible	4	4	4	5s	5s	5s	2.2s/-	1.9s/-
grep.55cf7b6a	Correct	Correct	Correct	2	2	2	3s	3s	3s	0.8s/-	0.4s/-
grep.3220317a	Plausible	Plausible	Plausible	27	27	27	41s	1m 5s	1m 11s	1.2s/-	1.1s/-
grep.db9d6340	Plausible	Plausible	Plausible	2	2	2	2s	3s	2s	1.8s/-	0.6s/-
grep.c96b0f2c	Plausible	Plausible	Plausible	41	41	41	1m 19s	1m 59s	2m 11s	2.2s/-	2.3s/-
grep.5fa8c7c9	Correct	Correct	Correct	34	34	34	55s	2m 43s	1m 35s	4.9s/-	3.8s/-
grep.54d55bba	-	-	Plausible	400	0	26	2m 42s	TO	2m 59s	4.1s/TO	2.4s/5.2s
Overall	10	10	13	107.2	14.8	19.2	1m 5s	1m 21s	1m 24s	2.2s/-	1.6s/4.9s

We conducted all experiments on an Intel® Core™ i7-2600 CPU 3.40GHz machine running Ubuntu 14.04 with 8GB of memory.

6.3.2 Program repair

To investigate the effect of second-order constraints on path explosion, we compared SE-ESOC configurations with Angelix (FO). Particularly, we executed repair algorithms described in Section 6.2.3 on the subjects of DBG-Bench, using developer-provided tests as the correctness criteria for patches.

For each suspicious program location, we bounded[2] the number of explored paths to 400, and used a 10 minutes time limit. SE-ESOC configurations used the languages of interpretations given in Figure 6.4a and Figure 6.4b. The same languages were used by Angelix (FO) synthesizer. Besides, we specified the depth bound $D \coloneqq 3$ for the synthesized functions.

Table 6.2 summarizes the results of our experiments. The column "Subject" lists subject program and their versions (commit hashes). The columns "Patch" show if a patch was generated by each configuration; we present only versions for which at least one configuration generated a repair. In these columns, "Correct" indicates that the generated patch is syntactically equivalent to the developer patch, otherwise the patch is classified as "Plausible".

In order to investigate the cause of failures of some configurations to find a patch, we collected additional statistics of symbolic execution sessions. For each configuration, we collected data for the session in which the program is executed symbolically with the failing test and a symbolic variable is installed in the fix location. Specifically, the columns "Paths" in Table 6.2 denote how many paths were explored by each configuration during this symbolic execution session. The columns "Time" show the time taken by each configuration to explore these paths.

Overall, the configurations based on second-order constraints generated all the patches generated by the approach based on first-order constraints, and SO/PSE also found three additional patches for find.091557f6, find.dbcb10e9

[2]It is common for synthesis-based program repair techniques to rely on path bounds. For instance, an enumerative approach SPR [62] uses the bound of 11 paths.

and grep.54d55bba. For each of these three cases, FO reached the limit of 400 paths during exploration, while SO/PSE reduced the number of explored paths to 14-26, which led to the successful generation of patches. However, SO/PSE required slightly more time on average for path exploration compared with FO (1m 24s for SO/PSE, and 1m 5s for FO).

In all three cases for which SO/PSE exclusively generated patches, the modified statements are executed multiple times by the failing tests. To explain how the reduction of explored paths is achieved in this case, we consider the experiments with the bug find.091557f6 in greater details. This bug in `find` utility is caused by a wrong handling of symbolic link loops when searching for files in a directory. One of the possible correct fixes is to add the disjunct `ent->fts_errno == ELOOP` to the condition shown in Figure 6.5a.

To synthesize a patch, Angelix (FO) replaces the buggy condition with a first-order variable, and performs symbolic execution to infer synthesis specification. The condition is evaluated multiple time during an execution of the failing test, and Angelix (FO) fails to infer sufficient specification to synthesize a patch due to the path explosion (since it terminates after reaching the limit of 400 paths).

Contrary to Angelix, SE-ESOC injects a second-order variable ρ applied to local program variables as shown in Figure 6.5a. It also associates the language in Figure 6.4a and the bound $D \coloneqq 3$ with ρ, that effectively define the search space of patches. This enables SE-ESOC to take advantage of the stronger notion of infeasibility (Definition 17) to prune irrelevant paths. Specifically, it determines that only 16 execution paths are feasible w.r.t. the considered language of interpretations, and synthesizes the interpretations $\rho_1, ..., \rho_{16}$ in Figure 6.5b corresponding to the 16 feasible paths.

In order to prune infeasible paths, SE-ESOC has to solve more complex constraints than FO, which results in additional performance overhead. However, the execution time of SE-ESOC is less dependent on the bounds of symbolic execution. To demonstrate this, we executed FO and SE/PSE with the example in Figure 6.5a, ranging the number of explored paths (`--max-forks` KLEE options) from 100 to 2000. The results of our experiment are presented in Figure 6.5c. As can be seen, the time taken by FO increases with the increase of the path bound, while the time taken by SE/PSE does not depend on the path bound, since it only has to explore 16 paths.

Overall, SE-ESOC helps alleviate path explosion when repaired expressions are executed *multiple times* during program execution.

6.3.3 Second-order solving

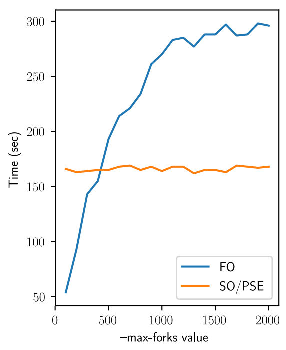
Since our approach relies on the notion of infeasibility (Definition 17) to prune explored paths, it is critical that it can efficiently handle unsatisfiable second-order queries. To investigate how existing component-based synthesis (CBS) encoding (Section 2.4.1) and the introduced propositional (PSE) encoding (Section 6.2.1) perform on unsatisfiable formulas that occur in the context of symbolic execution, we collected solver statistics in Table 6.2. The columns "SAT/UNSAT" demonstrate the average time taken by satisfiable and unsatisfiable queries during path exploration ("-" indicates that no such queries are performed). As can be seen, both CBS and PSE demonstrated comparable performance on satisfiable formulas (on average, CBS requires 2.2s, and PSE requires 1.6s). However, CBS did not solve any unsatisfiable queries within the time limit (10 minutes), while the average time taken by PSE on unsatisfiable queries is 4.9s, which is similar to the time taken by satisfiable queries. Thus, PSE is crucial in enabling SE-ESOC to avoid infeasible paths.

```
...
else if (ent->fts_info == FTS_DC)
  {
    issue_loop_warning(ent);
    error_severity(1);
    return;
  }
// ρ(ent->fts_info, ent->fts_errno, prev_depth)
else if (ent->fts_info == FTS_SLNONE)
  {
    if (symlink_loop(ent->fts_accpath))
      {
        error(0, ELOOP, ent->fts_path);
        error_severity(1);
        return;
...
```

(a) Buggy condition in find.091557f6.

ρ_1 :=(4 <= ent->fts_info)
ρ_2 :=!(ent->fts_errno == prev_depth)
ρ_3 :=((4 < ent->fts_info) && (prev_depth <= ent->fts_errno))
ρ_4 :=!(0 == ent->fts_errno)
ρ_5 :=((ent->fts_info == ent->fts_errno) || (9 <= prev_depth))
ρ_6 :=(ent->fts_info == (7 + prev_depth))
ρ_7 :=((prev_depth + ent->fts_errno) == (ent->fts_info - 6))
ρ_8 :=((ent->fts_errno < prev_depth) || (6 == ent->fts_info))
ρ_9 :=(ent->fts_info < (4 + ent->fts_errno))
ρ_{10} :=(ent->fts_info <= (ent->fts_errno + 6))
ρ_{11} :=!(ent->fts_info == 6)
ρ_{12} :=(0 <= prev_depth)
ρ_{13} :=!(4 < ent->fts_info)
ρ_{14} :=!(1 <= prev_depth)
ρ_{15} :=((ent->fts_errno < prev_depth) || (ent->fts_info <= 1))
ρ_{16} :=((ent->fts_errno < 32) || (prev_depth == ent->fts_info))

(b) Interpretations of ρ found along feasible paths.



(c) Time and explored paths.

Figure 6.5: Repairing wrong handling of symbolic link loops in `find`.

Chapter 7

Related work

The contributions of this thesis are related to several areas of research: program repair, program synthesis, debugging, symbolic execution, software transplantation and equivalence checking.

7.1 Program repair

Since tractability and precision are the main challenges of program repair, most of existing works are focused either on designing a better search algorithm to scale program repair to large programs and large search spaces, or on developing methods to alleviate the test overfitting problem. Although the techniques proposed in this thesis form a cohesive framework, they can also be divided based on the problem they address. Specifically, Angelix [70] that proposes an approach of inferring concise specification presented in Chapter 4 and symbolic execution with existential second-order constraints presented in Chapter 6 primarily tackle the tractability problem, while DirectFix [69] that proposes an approach of synthesizing minimal changes presented in Chapter 3 and SemGraft [67] that proposes an approach of using a reference implementation presented in Chapter 5 are designed to alleviate test overfitting and increase the quality of generated patches.

7.1.1 Addressing tractability

Existing approaches to address the tractability can be classified into two categories: syntactic and semantic.

Syntactic program repair techniques search for patches by generating and validating individual source code changes. GenProg [108] and JAFF [5] optimizes this process using genetic programming. Specifically, they apply mutation and crossover operators to create generations of program versions and evaluate them using a fitness function that counts the number of passing test. Subsequently, RSRepair [83] and AE [107] replace the genetic programming algorithm of GenProg with random search and adaptive repair search strategies, respectively. The benefit of these approaches is that they scale to large programs such as PHP and Wireshark [56]. However, a recent study [85] revealed that the quality of the repairs generated from these tools might be unsatisfactory — a large number of these repairs simply delete functionality.

Contrary to syntactic techniques, semantic approaches aim to identify the meaning of defects by means of semantic analysis. Particularly, they analyze the behaviour of the program to infer a specification (expressed using e.g. logical formulas or concrete values) that is used to synthesize a patch.

SemFix infers a patch specification using first-order symbolic execution and synthesizes a patch using SMT-based program synthesis as described in

Section 2.6.2. The techniques introduced in this thesis follow the general workflow of SemFix [73], but they substantially extend it. Specifically, the introduced techniques (1) help to scale semantic repair to large programs by inferring a concise value-based specification (Chapter 4), (2) alleviate the path explosion problem by using symbolic execution with existential second-order constraints (Chapter 6), (3) the proposed techniques help to increase the quality of generated patches by synthesizing minimal modifications (Chapter 3) and by inferring the missing specification from a reference implementation (Chapter 5).

Besides SemFix, other semantic approaches include Nopol [113], SPR [62] and an approach by Nguyen at el. [74]. Nopol [113] and SPR [62] differ from SemFix in that they infer a more simple synthesis specification based on angelic values. However, since these approach infer this specification by enumerating and testing sequences of values, they are limited to repairing boolean expressions (e.g. buggy if- and loop-conditions). Meanwhile, the technique of inferring a value-based specification (angelic forest) described in Chapter 4 utilizes symbolic execution and therefore can also be used to repair integer expressions and synthesize multi-line changes. Similarly to SemFix, Nopol and SPR also suffer from the path explosion. For instance, SPR bounds the number of explored paths to 11, which may lead to inferring insufficient specification and failing to synthesize a patch. Potentially, the technique for alleviating path explosion described in Chapter 6 can be adapted for Nopol and SPR.

Nguyen at el. [74] demonstrated that program repair can be naturally encoded as a reachability problem. However, this approach has limitations:

(1) it requires encoding several tests inside a single meta-program, which is non-trivial for real-world applications because of complex execution setup and the presence of global state, and (2) it is restricted to relatively simple patch templates. Our methodology addresses the above limitations.

7.1.2 Addressing precision

Since a test-suite is an incomplete specification, test-based repair approaches may generate patches that do not correspond to user intentions but merely overfit the tests [95]. To address this problem, several general approaches has been investigated: designing a better search space, prioritizing patches, test generation and inferring additional specification.

To design a better search space, PAR [48] uses manually selected human patch templates, which leads to the increase of repair quality. Genesis [61] extends the approach of PAR by inferring patch templates automatically from history. These approaches are orthogonal to the techniques described in this thesis.

To prioritize high quality patches, various heuristics have been proposed. In Chapter 3, we introduce an approach that synthesizes syntactically minimal changes using a MAX-SAT solver. The intuition of this approach is that smaller changes are easier to understand for users and they break less unspecified functionality. Qlose [25] extends this approach by prioritizing changes based not only on syntactic, but also on semantic distance. Prophet [63] applies machine learning in order to learn universal properties of human patches, and uses the learned model to prioritize patches in its search space.

To repair to given defect, it searches for a patch that passes the tests and also is the most similar to human patches according to the learned model. Other prioritization strategies include learning from history [55], using comments and mining common predication [112], combining several syntactic and semantic metrics [54], etc.

Another approach of addressing test overfitting is to augment existing test suites by improving e.g. test suite metrics [116]. Test suites can be augmented using existing testing techniques such as search-based test generation [110] or fuzzing [114].

In Chapter 5, we proposed an approach of improving the quality of automatically generated patches by inferring missing specification from a reference implementation. Although this approach can be applied on when a reference implementation is available, its advantage over previous techniques to address test overfitting consists in stronger correctness guarantees.

7.1.3 Domain specific repair

Apart from general-purpose repair algorithms like ones introduced in this thesis, there are also other repair approaches targeted for specific types of defects (e.g., buffer overflow) or specific application domains (e.g., web applications) [78, 39, 14, 22, 94, 90, 75, 88, 32]. Also, many previous works assume the existence of formal specification or contracts [35, 40, 34, 50, 87, 26, 106] unlike test-driven approaches such as ours. Relifix [100] utilizes previous program versions in order to perform automated repair of regression bugs, however it relies on syntactic similarity of the previous and the buggy programs, which distinguishes it from our approach. Lastly, MintHint [45] suggests a repair hint instead of a patch by allowing some tests to remain failing and thereby performing statistical analysis for a class of semi-repairs satisfying this relaxed requirement.

7.2 Program synthesis

Existing program synthesis techniques can be used to generate patches by directly searching in patch spaces, however this approach has limitations because of the complexity of repaired programs. For instance, Sketch [96] fills "holes" in sketches (partial programs), and can be potentially applied to generate repairs for identified suspicious statements. However, it translates programs into boolean formulas and therefore can repair only relatively small programs [36]. Since program synthesis algorithms may not be directly applicable to program repair, they are used as parts of program repair algorithms for filling "holes" in programs based on inferred specification.

To synthesize patches based on inferred specification, early semantic techniques such as SemFix and Nopol relied on component-based synthesis proposed by Jha et al. [38]. In principle, any syntax-guided synthesis algorithm [3] can be used for performing this task. However, symbolic execution with existential second-order constraints introduced in Chapter 6 for addressing path explosion problem poses additional requirement on the underlying program synthesis algorithms. Enumerative techniques [3, 4] that explicitly generate and test individual terms cannot be applied in the context of symbolic execution because they would require checking satisfiability of path constraint for each possible expression in the search space. The algorithm of Reynolds et al. [86] implemented inside an SMT solver tends to synthesize complex solutions consisting of thousands of nodes[1] that cannot be understood by humans, which makes it unsuitable for program repair. The encoding of second-order formulas proposed by Jha et al. [38] relies on linear integer arithmetic constraints, which results in inefficient proofs of unsatisfiability. On the other hand, symbolic execution requires checking unsatisfiability of path constraints to avoid infeasible paths, therefore its performance critically depends of the efficiency of unsatisfiability proofs. The propositional second-order formula encoding introduced in Chapter 6 addresses the above limitations of existing techniques.

7.3 Debugging

The purpose of debugging is to identify program statement or locations responsible for the bug. Thus, debugging can be considered as one of the steps of program repair.

Statistical bug isolation [41] identifies suspicious program statements using statistical information that consists in the number of times each statement is executed by passing and failing test cases. The described program repair system relies on statistical bug isolation, since such approaches demonstrate good scalability in practice.

Angelic debugging [15] aims to discover an angelic value for a program expression, that is a value that would enable the given buggy program to pass

[1]SyGuS-Comp 2017 results: `http://sygus.seas.upenn.edu/SyGuS-COMP2017.html`

the failing test. In order to discover such values, angelic debugging replaces the considered expression with a symbolic variable, and applied symbolic execution to find a path along which the program would pass the test. Then, a angelic value can be obtained by solving the corresponding path condition. SemFix differs from angelic debugging in that it synthesizes new expressions rather than finds angelic values. Angelix (Chapter 4) extends the idea of angelic value to angelic forest — a concise specification that can capture the effect of complex multi-line changes.

BugAssist [43] exploits MAX-SAT for fault localization. Specifically, it encodes the fault localization problem as a Partial MAX-SAT problem, where the semantics of the program statements is encoded as soft constraints, and the tests are encoded as hard constraints. Then, a Partial MAX-SAT solver finds the minimal number of clauses that need to be relaxed to make the formula satisfiable, corresponding to the minimal number of statements that need to be modified to enable the program to pass the tests. However, unlike the approach in Chapter 3, MAX-SAT is not used for repair synthesis — BugAssist does not suggest changes for the localized statements.

Darwin [82] is a semantic fault localization approach that uses a correct previous version (or a reference version) of the analyzed software to precisely identify faulty statement. Specifically, for a given failing test, it symbolically analyzes the buggy and the correct versions and generates a new test that follows a different path in the buggy program compared with the original test, and the same path in the correct version as the original test. Then, this new test is used to analyze the differences in the behaviours of the two programs and to localize the statements responsible for these differences.

SemGraft [67] introduced in Chapter 5 also uses a correct program version, however it extracts additional specification in order to improve the quality of automatically generated patches.

7.4 Symbolic execution

Symbolic execution has been originally proposed for software testing [49, 13], but has since found a wide range of other applications in software engineering including software verification [37], program debugging [82] and program repair [73]. This thesis studies an application of symbolic execution to program repair (Chapter 4) and conditional equivalence checking (Chapter 5), and addresses a fundamental limitation of symbolic execution — the path explosion problem (Chapter 6). Contrary to the applications in software testing, symbolic execution is used by the introduced techniques as a problem comprehension mechanism (to extract specification from tests (Chapter 4) or a reference implementation (Chapter 5), rather then an analysis/bugfinding mechanism. Overall, this thesis demonstrates that symbolic execution is effective in areas beyond software testing.

Godefroid [33] proposed to use higher order constraints to model imprecision of symbolic execution. Specifically, this approach replaces unknown/-complex instructions with uninterpreted functions and generates inputs by solving *universal* constraints over *uninterpreted* functions. The main difference of our approach introduced in Chapter 6 is that it solves *existential* constraints over functions whose interpretations are restricted by a user-defined language, which implies different methodology and applications. Specifically, the approach with uninterpreted functions cannot reduce the number of explored paths as shown in Section 6.1.3 in the context of program repair, since this reduction is achieved by restricting of the space of interpretations. Palikareva et al. [76] proposed to test divergences between program versions by encoding two versions in a single symbolic execution session. Our approach differs in that it encodes a potentially infinite number of program versions inside a single symbolic execution session.

7.5 Software transplantation

Automated software transplantation [7, 80] tools like μScalpel [8] and CodeCarbonCopy [91] aim at transplanting new functionality from a donor application into a recipient application. CodePhage [92] can fix program errors like out of bounds access, integer overflow, and divide by zero errors. They focus on finding an error checking code in the donor application that can serve as a compensation for a missing check in the host application. Since their approach tries to copy code, it is necessary to translate the check from the data structures and name space of the donor into an application-independent representation suitable for insertion into the recipient application. The advantage of SearchRepair [47] compared to other repair approaches is the use of semantic code search [98, 99] to identify suitable code fragments for repair. Our approach differs from software transplantation literature, since we do not copy or transplant any functionality from a donor program. Instead, we synthesize functionality to meet a correctness criteria — either to pass given tests or to enforce the equivalence with the reference implementation. Although, SemGraft presented in Chapter 5 also uses a reference/donor program, it uses it for a different purpose, namely for inferring missing specification. We also differ from recent works on grafting of code clones [117], since this line of work seeks to achieve greater test-reuse across code clones for the sake of differential testing.

7.6 Equivalence checking

Lahiri et al. [51] proposed to find the rootcause for equivalence failures by leveraging semantic similarity between two program binaries. Since they aim to extract a complete specification, their approach scales only to small programs. SemGraft [67] introduced in Chapter 5 sacrifices completeness for the sake of applicability by checking conditional equivalence of a buggy program and a reference program w.r.t. a user-defined input condition.

Chapter 8

Discussion

Debugging is notoriously difficult and time consuming. Automated program repair is a promising technology that can reduce the burden of debugging by automatically fixing defects. Early program repair techniques utilized syntactic search in a space of patches. Although such techniques demonstrated encouraging results, they suffer from effectiveness and quality limitations. In the meanwhile, semantic-based repair that comprehends the meaning of the program and the defect scaled to only relatively small programs. We develop a series of semantic analysis techniques that improve the quality of automatically generated patches and scale to large real-world software.

8.1 Summary of contributions

The contributions of the thesis are the following:

- We propose an approach of encoding the repair problem as a instance of maximum satisfiability problem by reusing existing program synthesis and error diagnosis methods (Chapter 3).

- We devise a concise semantic signature that scales constraint-based repair to large real-world programs and that is capable of representing complex program changes (Chapter 4).

- We suggest an approach to improve the quality of automatically generated patches by inferring missing specification from a reference implementation (Chapter 5).

- We introduce symbolic execution with existential second-order constraints that helps to alleviate the path explosion problem of traditional symbolic execution in the context of program repair (Chapter 6).

8.2 Perspectives

In this section, we draw conclusions from the results of this thesis, and also discuss future research directions.

8.2.1 Efficient patch generation

Ideally, a program repair system should be efficient, effective and precise. However, it might be infeasible to achieve all these qualities simultaneously. For example, by using more heavy-weight methods that check for conditional equivalence of the buggy and the reference programs (Chapter 5), it is possible to increase the quality of generated patches, however this would also impose additional overhead and require manually specifying input conditions for equivalence checking. Similarly, symbolic execution with existential second-order constraints (Chapter 6) helps to repair more defects in loops, however is less efficient for the cases without loops compared with traditional first-order symbolic execution because of the overhead of second-order constraint solving. Thus, when designing a program repair system, it is important to balance the trade-offs between its efficiency, effectiveness and precision.

In this section, we demonstrate how to simplify and extend the proposed algorithms to design a program repair system that, although does not provide high correctness guarantees like SemGraft (Chapter 5), yields an order of magnitude efficiency improvement over existing repair techniques. In order to achieve this, we propose a methodology based on a test-equivalence relation [44, 53]. If two programs are test-equivalent for a test, then the programs produce indistinguishable results on that test:

Definition 19 (Test-equivalence)**.** *Let $\mathcal{P}$ be a set of programs, t be a test. An equivalence relation (reflexive, symmetric and transitive) $\overset{\mathrm{t}}{\sim} \subset \mathcal{P} \times \mathcal{P}$ is a* test-equivalence relation *for t if it is consistent with the results of t, that is $\forall p_1, p_2 \in \mathcal{P}$, if $p_1 \overset{\mathrm{t}}{\sim} p_2$ then p_1 and p_2 either both pass t or both fail t.*

The proposed algorithm partitions the space of candidate patches into *test-equivalence* classes by performing on-the-fly analysis during test execution, which distinguishes it from a previous work [107] that relies on *program equivalence.* Compared with syntax-based techniques, it reduces the number of test executions since a single execution is sufficient to evaluate multiple patch candidates (specifically, all patches in the same test-equivalence class). Compared with techniques based on path exploration (e.g. SemFix [73] and SPR [62]/Prophet [63]), it reduces the number of test executions for two reasons. First, similarly to symbolic execution with existential second-order constraints (Chapter 6), it avoids exploration of "infeasible" paths (sequences of values), i.e. paths or sequences of values that cannot be induced by any of the considered candidate patches in the context of given tests. Secondly, it reuses information inferred across multiple tests to skip redundant executions, while previous semantics-based techniques perform path exploration independently for each test.

Consider a defect in the revision 0661f81 of Libtiff[1] from the GenProg ICSE'12 benchmark. The code in Figure 8.1a is responsible for flushing data written by the compression algorithm, and the defect is caused by the wrong highlighted condition. Libtiff test-suite contains 78 tests, and this defect is manifested by a failing test called "tiffcp-split". Figure 8.1b demonstrates the developer patch that modifies the wrong condition by removing the clause `tif->tif_rawcc != orig_rawcc`.

We demonstrate how existing automated program repair algorithms generate a patch for this condition. First, repair algorithms perform fault localization to identify suspicious program statements. The number of localized statements in existing tools may vary from tens to thousands depending on algorithms and configurations (it can potentially include all executed statements). In this example, we consider only the location of the buggy expression highlighted in Figure 8.1a.

Second, program repair algorithms define a *search space* of candidate patches. In this work, we primarily focus on two state-of-the-art approaches that have been shown to scale to large real-world programs: Angelix [70] and

[1]Libtiff is a software library that provides support for TIFF image format: `http://simplesystems.org/libtiff/`

```
...
(*tif->tif_close)(tif);
if (tif->tif_rawcc > 0
    && tif->tif_rawcc != orig_rawcc
    && (tif->tif_flags & TIFF_BEENWRITING)!= 0
    && !TIFFFlushData1(tif)) {
      TIFFErrorExt(tif->tif_clientdata,
        module,
        "Error␣flushing␣data␣before␣directory␣write");
    return (0);
  }
...
```

(a) Incorrect condition in Libtiff (rev. 0661f81).

```
...
(*tif->tif_close)(tif);
if (tif->tif_rawcc > 0
    && (tif->tif_flags & TIFF_BEENWRITING)!= 0
    && !TIFFFlushData1(tif)) {
      TIFFErrorExt(tif->tif_clientdata,
        module,
        "Error␣flushing␣data␣before␣directory␣write");
    return (0);
  }
...
```

(b) Developer patch for incorrect condition.

Figure 8.1: Defect in Libtiff library from GenProg ICSE'12 benchmark.

Prophet [63]. Specifically, our goal was to support a combination of transformations implemented in these systems. Thus, the search space for the highlighted condition includes all possible replacements of its subexpressions by expressions constructed from visible program variables and C operators, refinements (e.g. appending `&& EXPR` and `|| EXPR`), replacements of operators and swapping arguments. In total, the search space in our synthesizer contains 56 243 modifications of the buggy condition.

Finally, program repair algorithms explore the search space in order to try to find a modification that passes all given tests. We say that an element of a search space is *explored* if the algorithm identifies if it passes all the tests or fails at least one. Existing search space exploration methods can be classified into two categories: syntax-based and semantics-based. *Syntax-based* algorithms explicitly generate and test syntactic changes. In this example, a syntax-based algorithm have to execute the failing test 56 243 times to evaluate all candidates[2]. Since there are 78 tests in the test-suite, 907 457 test executions are required to explore the search space[3]. Given the high cost of test execution, this approach has poor scalability.

Semantics-based techniques (e.g. Semfix [73], SPR [62], Angelix and Prophet) split exploration into two phases. First, they infer a synthesis specification for the identified expression through path exploration. For this example, they enumerate and execute sequences of condition values (e.g. $\mathit{true}, \mathit{true}, \mathit{true}, \mathit{false}, ...$) to find those sequences that enable the program to pass the test. Second, they synthesize a modification of the condition to match the inferred specification. In this example, there are 256 possible execution paths (the condition is evaluated multiple times during the test execution), therefore a semantics-based algorithm performs 256 test executions for the failing tests, and 1320 for the whole test-suite[4]. Although semantics-based techniques were shown to be more scalable [62], they are subject to the *path explosion problem*: the number of execution paths can be infinite. To address this, current systems introduce a bound for the number of explored paths, however it may affect their effectiveness: if a path followed by the

[2]Since the search space contains the correct patch in this example, the algorithm can stop search earlier after the patch is found. Then, the number of test executions depends on the exploration order.

[3]This data is obtained by executing our implementation of syntactic enumeration.

[4]This data is obtained by executing Angelix.

```
1. ((tif->tif_rawcc > 0) && (tif->tif_rawcc != orig_rawcc))
    || (tif->tif_flags & TIFF_BEENWRITING)
2. ((tif->tif_rawcc > 0) || (tif->tif_rawcc != orig_rawcc))
    && (tif->tif_flags & TIFF_BEENWRITING)
3. ((tif->tif_rawcc == 0) && (tif->tif_rawcc != orig_rawcc))
    && (tif->tif_flags & TIFF_BEENWRITING)
4. (((tif->tif_rawcc > 0) && (tif->tif_rawcc != orig_rawcc))
    && (tif->tif_flags & TIFF_BEENWRITING)) || (imagedone >= orig_rawcc)
5. (((tif->tif_rawcc > 0) && (tif->tif_rawcc != orig_rawcc))
    && (tif->tif_flags & TIFF_BEENWRITING)) || (tif->tif_flags >= 74)
```

Figure 8.2: 5 expressions representing different test-equivalence classes.

correct patch is omitted, then this correct patch cannot be generated.

The algorithm proposed in this work performs on-the-fly partitioning of program modifications into test-equivalence classes. We demonstrate the effect of the relation $\overset{t}{\sim}_{value}$. Two modifications of a program expression are test-equivalent w.r.t. $\overset{t}{\sim}_{value}$ if they are evaluated into the same sequences of values during the test execution. Surprisingly, the space of 56 243 modifications can be partitioning into only 5 test-equivalence classes for the failing test "tiffcp-split" w.r.t. $\overset{t}{\sim}_{value}$; five elements of the search space that represent different test-equivalence classes are given in Figure 8.2. Since all patches in the same test-equivalence class exhibit the same behaviour for the corresponding test, the failing test can be executed only 5 times to evaluate all candidates.

Our algorithm computes test-equivalence classes for each test in the test-suite. However, since test-equivalence classes for different tests may intersect, our algorithm takes advantage of this to skip redundant execution across different tests. Specifically, for each next test it only evaluates subspaces of modifications that are not included into failing test-equivalence classes of previously executed tests. Meanwhile, semantics-based techniques perform specification inference for each test independently without reusing information across tests. As a result, our algorithm requires only 103 test executions to evaluate all 56 243 modifications with the whole test-suite.

The key insight that enables our method to reduce the number of required test executions is that, compared with techniques that explore execution paths, it takes the expressiveness of the patch space into account (e.g. it identifies that only 5 out of 256 possible execution paths are induced by the considered set of 56 243 transformations). Compared with syntactic enumeration, it substantially reduces executions since a single execution evaluates a whole test-equivalence class.

The described algorithm can be considered as a special case of symbolic execution with existential second-order constraints (Chapter 6), that relies on values rather that path conditions to partition the space of paths, and therefore cannot take program dependencies into account. However, it also imposes lower performance overhead since it does not require solving second-order queries.

Since a test-suite is an incomplete specification, test-driven program repair suffers from the test overfitting problem [95]. To address this issue, state-of-the-art techniques define a priority (a cost function) in the space of patches and search for a program modification that optimizes this function. Ideally, this function should assign higher cost to overfitting patches. For instance, Prophet [63] demonstrates how such a cost function learned from human patches enables the generation of more correct repairs.

Consider a program p in Figure 8.3a that counts odd numbers in the interval $(0, i]$. The `*` indicates a wrong condition that has to be modified by the repair algorithm (the correct condition is `i mod 2` = `1`). We denote a program obtained by substituting $*$ with an expression e as $p[{}^{*}\!/_{e}]$. The repair algorithm searches for a plausible patch (a substitution of $*$ with a condition) from the space $\mathcal{P}$ in Figure 8.3b such that the resulting program passes the test t defined as follows:

$$t \coloneqq (\{\ i \mapsto 4,\ c \mapsto 0\ \},\ \lambda\sigma.\ \sigma(c) = 2)$$

where t is pair of (1) an initial program state (mapping from variables to values) and (2) a test assertion (a boolean function over program states) denoted using lambda notation. We assume that $*$ is such that p fails t. Besides that, we consider a cost function κ defined for the considered space of substitutions in Figure 8.3c. The goal is to find a plausible patch with the lowest cost.

In order to find a patch for the example program, techniques like Angelix and Prophet enumerate possible sequences of values that a condition can take during test execution. Since there can be potentially infinite number of such sequences, existing approaches introduce a bound for the number of explored sequences and use an exploration heuristics to choose which sequences to explore. For instance, Prophet enumerates sequences where the condition first always takes the true branch until a certain point after which it always takes the false branch. Thus, for the considered example it would enumerate the following sequences:

$$\{\ \mathit{true}, \mathit{true}, \mathit{true}, \mathit{true}\ \},$$
$$\{\ \mathit{true}, \mathit{true}, \mathit{true}, \mathit{false}\ \},$$
$$\{\ \mathit{true}, \mathit{true}, \mathit{false}, \mathit{false}\ \},$$
$$\{\ \mathit{true}, \mathit{false}, \mathit{false}, \mathit{false}\ \}$$

For each of these sequences, Prophet executes the program with the test t in such a way that the condition `*` takes the values as in this sequence during the execution. Only the third sequence $\{\ \mathit{true}, \mathit{true}, \mathit{false}, \mathit{false}\ \}$ enables the program to pass t, therefore it will be selected as a specification for expression synthesis. The synthesizer will find the expression $\mathtt{i} > \mathtt{2}$ obtaining a suboptimal patch $p[{}^{*}/_{\mathtt{i}\,>\,\mathtt{2}}]$ with the cost 0.5, since this is the only expression from the search space satisfying the specification. However, the correct expression $\mathtt{i}\ \mathtt{mod}\ \mathtt{2} = \mathtt{1}$ with a lower cost 0.3 cannot be generated, since the corresponding sequence $\{\ \mathit{false}, \mathit{true}, \mathit{false}, \mathit{true}\ \}$ is not explored by the algorithm.

In contrast to techniques like Angelix and Prophet, our algorithm iterates through the search space in such a way that at each steps it selects and evaluates an unevaluated candidate with the lowest cost. Specifically, it starts by choosing the candidate $p[{}^{*}/_{\mathtt{i}\,\geq\,\mathtt{0}}]$ with the cost 0.1. It executes this candidate on-the-fly computing its test-equivalence class w.r.t. $\overset{\mathtt{t}}{\sim}_{\mathit{value}}$. This class contains the program $p[{}^{*}/_{\mathtt{c}\,\geq\,\mathtt{0}}]$, since the conditions $\mathtt{i} \geq \mathtt{0}$ and $\mathtt{c} \geq \mathtt{0}$ produce the same sequence of values $\{\ \mathit{true}, \mathit{true}, \mathit{true}, \mathit{true}\ \}$ for t. Since $p[{}^{*}/_{\mathtt{i}\,\geq\,\mathtt{0}}]$ does not pass the test, the whole corresponding test-equivalence

```
while i > 0 do
  if * then
    c := c + 1
  fi;
  i := i - 1
od
```

(a) Buggy program p.

$$\begin{aligned}\mathcal{P} := \{\ & p[{}^{*}/_{\texttt{i} \geq \texttt{0}}],\\ & p[{}^{*}/_{\texttt{c} \geq \texttt{0}}],\\ & p[{}^{*}/_{\texttt{i mod 2} = \texttt{1}}],\\ & p[{}^{*}/_{\texttt{i mod 2} = \texttt{0}}],\\ & p[{}^{*}/_{\texttt{i} > \texttt{2}}]\ \}\end{aligned}$$

(b) Search space.

$$\begin{aligned}\kappa(p[{}^{*}/_{\texttt{i} \geq \texttt{0}}]) &:= 0.1\\ \kappa(p[{}^{*}/_{\texttt{c} \geq \texttt{0}}]) &:= 0.2\\ \kappa(p[{}^{*}/_{\texttt{i mod 2} = \texttt{1}}]) &:= 0.3\\ \kappa(p[{}^{*}/_{\texttt{i mod 2} = \texttt{0}}]) &:= 0.4\\ \kappa(p[{}^{*}/_{\texttt{i} > \texttt{2}}]) &:= 0.5\end{aligned}$$

(c) Cost function.

Figure 8.3: Example of optimal program repair problem.

class is marked as failing. Next, it selects $p[{}^{*}/_{\texttt{i mod 2} = \texttt{1}}]$ with the cost 0.3 since $p[{}^{*}/_{\texttt{c} \geq \texttt{0}}]$ was indirectly evaluated through test-equivalence at the previous step. Since this candidate passes the test, the algorithm outputs it as a found repair.

Our algorithm guides exploration based on a given cost function and focuses on high priority areas of the space of patches. By construction, if it finds a patch, then this patch is guaranteed to be the global optimum in the search space w.r.t. the cost function. Angelix and Prophet, on the other hand, may spend executions for value sequences that correspond to suboptimal candidates or correspond to no candidates at all (e.g. $\{$ *false*, *true*, *true*, *true* $\}$), and therefore may miss the best patch in their search space.

Although current program repair approaches have been shown to be relatively effective in modifying existing program expressions, they provide limited support for more complex transformations. We consider one such transformation that inserts a new assignment statement to the buggy program. Techniques like Prophet and GenProg can generate patches by copying/moving existing program assignments, however this approach has limitations: (1)

```
...
clear_bufs();
to_stdout = 1;
part_nb = 0;
ifd = part_nb;
if (decompress) {
  method=get_method(ifd);
...
```

(a) Before if-statement.

```
...
clear_bufs();
to_stdout = 1;
part_nb = 0;
ifd = 0;
if (decompress) {
  method=get_method(ifd);
...
```

(b) Before if-statement.

```
...
clear_bufs();
to_stdout = 1;
part_nb = 0;
if (decompress) {
  ifd = part_nb;
  method=get_method(ifd);
...
```

(c) Inside if-statement.

Figure 8.4: Candidate patches for defect of Gzip from GenProg ICSE'12 benchmark.

assignments for local variables cannot be copied from different parts of the program because of their scope and (2) each insertion of an assignment is validated separately, which yield a large number of required test executions. Existing techniques do not apply specification inference for assignment synthesis because such specification has to encode all possible side effects that can be caused by assignment insertion (for each variable that can appear in the left-hand side of the assignment), which makes inferring such specification infeasible for large programs.

We show how test-equivalence can scale assignment synthesis for a defect in Gzip[5] from the GenProg ICSE'12 benchmark. Consider three candidate patches in Figure 8.4 that insert the highlighted statements at several program locations. First, our algorithm identifies that the program in Figure 8.4a is test-equivalent to the program in Figure 8.4b (w.r.t. the relation $\overset{t}{\sim}_{value}$) since they differ only in the right-hand side of the highlighted assignments and the corresponding expressions take the same values during test execution. Second, using a simple dynamic dependency analysis our al-

[5]Gzip is a file compression/decompression application: `https://www.gnu.org/software/gzip/`

gorithm identifies that the program in Figure 8.4a is test-equivalent to the program in Figure 8.4c since (1) they insert the same assignment at different program locations, (2) both these locations are executed by the test since the true branch of the if-statement is taken during the test execution and (3) the variables `ifd` and `part_nb` are not used/modified between these locations during test execution. We refer to such a test-equivalence relation as $\overset{t}{\sim}_{deps}$. Finally, our algorithm merges the results of the two analyses (as the transitive closure of their union) and determines that the program in Figure 8.4b is test-equivalent to the program in Figure 8.4c. Therefore, a single test execution is sufficient to evaluate all these patches.

Since test-equivalence is a weaker property than the property of "passing the test" expressed by the inferred specification in semantics-based techniques, it permits using more lightweight analysis techniques. Specifically, we demonstrate that a composition of two lightweight test-equivalence analyses enables us to scale assignment synthesis.

We implemented the described approach in a tool called f1x. f1x combines the transformation schemas of SPR/Prophet and Angelix (we studied implementation of these systems in order to closely reproduce their search spaces). More details about the implementation and the algorithms are available in our technical report [68].

We evaluated the discussed methodology on the GenProg ICSE'12 benchmark [56] (with test suites independently augmented to prevent repair tools from generating implausible patches [85]). We selected the following systems and their configurations for evaluation:

F1X f1x that implements the test-equivalence partitioning technique.

F1X^E f1x^E is a variant of f1x that enumerates changes without test-equivalence partitioning. This variant is considered to evaluate implementation-independent effect of partitioning.

ANG Angelix 1.1 [70] that implements a symbolic path exploration and prioritizes syntactically small changes.

PR Prophet 0.1 [63] that implements value search (a variant of path exploration) for conditional expressions and patch prioritization based on machine learning.

PR* Prophet* that is a variant of Prophet that disables transformations for (1) inserting overfitting return insertions and (2) copying complex statements except for assignments. This variant is considered to match the transformation implemented in F1X/F1X^E, since the search space of F1X/F1X^E is effectively the combination of the search spaces of PR* and ANG.

GP GenProg-AE 3.0 [107] that implements a group of analysis techniques to avoid evaluating functionally-equivalent patches (as opposite to test-equivalent as in our approach).

We reuse the configurations from previous studies for running Angelix, Prophet and GenProg-AE [85, 107]. As Prophet takes a correctness model as input to prioritizes patches akin to the provided model, we used the default model that is publicly available[6].

[6]Prophet website: `http://rhino.csail.mit.edu/prophet-rep/`

Table 8.1: Effectiveness of program repair approaches.

Subject	Plausible						Equivalent to human					
	F1X	F1X^E	ANG	PR	PR*	GP	F1X	F1X^E	ANG	PR	PR*	GP
libtiff	13	10	10	5	3	5	5	3	3	2	1	0
lighttpd	5	3	-	4	4	4	0	0	-	0	0	0
php	15	7	10	18	15	7	6	3	4	10	6	2
gmp	2	1	2	2	2	1	2	1	2	1	1	0
gzip	3	2	2	2	2	2	2	0	1	1	1	0
python	5	1	-	6	5	3	0	0	-	0	0	1
wireshark	4	4	4	4	4	4	0	0	0	0	0	0
fbc	1	1	-	1	1	1	1	1	-	1	1	0
Overall	49	29	28	42	36	27	16	8	10	15	10	3

We conduct all experiments on Intel® Xeon™ CPU E5-2660 machines running Ubuntu 14.04, and use a 10 hours timeout for running each configuration.

Table 8.1 summarizes the effectiveness results for F1X, F1X^E, ANG, PR, PR* and GP. The second through seventh columns denote the number of plausible patches generated by each repair approach, while the eighth through thirteenth columns represent the number of patches syntactically equivalent to the human patches. As Angelix does not support lighttpd, python and fbc, the corresponding cells for these subjects are marked with "-". The overall results illustrate that F1X generates the highest number of plausible patches compared to all other evaluated repair approaches. The "Equivalent to human" column in table 8.1 shows that F1X generates 8 more human-like patches than F1X^E, 6 more human-like patches than ANG, 1 more human-like patch than PR, 2 more human-like patches than PR* and 13 more human-like patches than GP. All F1X patches are included in the anonymous supplementary material submitted with this paper.

We attribute the high number of patches generated by F1X to the larger

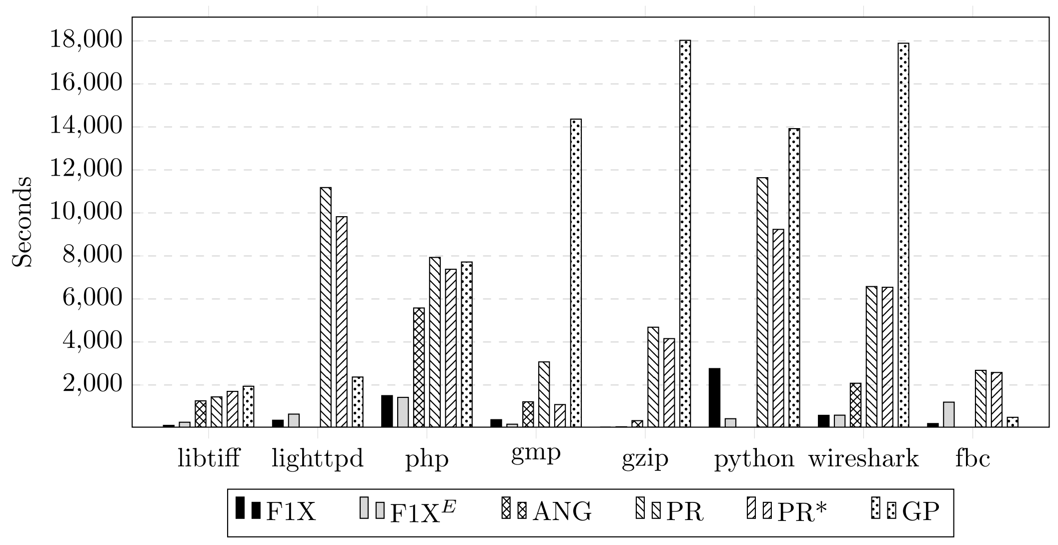


Figure 8.5: Average patch generation time.

patch space supported by F1X compared to other approaches. Since F1X combines the search spaces of ANG and PR*, it fixes all defects that are fixed by either of these tools. Note that F1X finds more patches than F1X^E within the time limit due to the performance gain from our partitioning.

Figure 8.5 illustrates the average patch generation time for the configurations. The x-axis of Figure 8.5 represents the eight subjects in the benchmark, while the y-axis shows the average time taken to generate a patch for all defects for a given subject where each bar depicts a patch generation approach. Overall, the average patch generation time for F1X is significantly shorter than all other repair approaches. For instance, F1X requires only 121 seconds on average to generate a patch for libtiff, while ANG takes 1262 seconds (F1X is $\frac{1262}{121}$=10.5X faster than ANG). Meanwhile, PR* takes 1701 seconds on average to produce a patch for libtiff (F1X is $\frac{1701}{121}$=14X faster than PR*). Notably, F1X is 16X faster than GP for libtiff (GP takes 1940 seconds on average to generate a patch for libtiff). The average patch generation time for PR is slightly higher compared to PR* as it searches through a slightly larger patch space.

The results shown in Figure 8.5 validate our claim that F1X is able to achieve significant improvement on the patch generation time due to its efficient search algorithm. F1X and F1X^E demonstrate a comparable average time of patch generation because F1X^E finds a subset of patches found by F1X that appears early in the sequence of explored candidates.

In future research, the trade-off represented by different techniques should be investigated to design a practical repair systems that would be able to address a large number of defects and also provide satisfactory precision.

8.2.2 Addressing test overfitting

Low precision is a fundamental limitation of existing program repair system, and it is primarily caused by the lack of specification in real-world software. Indeed, as has been shown in previous work [95], the use of tests as correctness criteria for program repair often leads to the generation of incorrect patches that merely overfit the tests.

In this thesis, we proposed two methods of addressing test overfitting. First, this problem can be alleviated by searching for a patch that maximally preserves the original source code (Chapter 3). Secondly, the missing specification can be automatically inferred from a reference implementation if it is available (Chapter 5), which can be used to improve the quality of generated patches and provide partial correctness guarantees.

In future research, we plan to investigate practical ways of discovering the missing specification and enforcing in a scalable manner. For instance, semantic code search [99] might be used to find suitable reference programs automatically, and used them to guide patch generation as in SemGraft (Chapter 5). Another promising research direction consists in exploiting the behavior similarity of test case executions as proposed by Xiong et al. [111].

8.2.3 Future application of program repair

Automated program repair targets the arising problem of low software quality, and can have a significant economic impact on software development. Specifically, we envision the following important future applications of automated program repair:

Programming environment Program repair might be integrated into programming environment to automatically suggest fixes for program defects during development/maintenance. This would help to reduce the cost of development/maintenance and also improve software quality.

Security Program repair might help to promptly address security vulnerabilities by automatically fixing security bugs. For example, we show that Angelix (Chapter 4) can automatically fix the infamous Heartbleed bug in OpenSSL[7].

Education Program repair might help to automate computer science education [115]. Specifically, it might be used for automatically providing feedback or automatically grading students assignments.

[7]Heartbleed bug: `http://heartbleed.com/`

Bibliography

[1] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A parametric approach for smaller and better encodings of cardinality constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 80–96. Springer, 2013.

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *Pacific Rim International Symposium on Dependable Computing*, pages 39–46. IEEE, 2006.

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design*, pages 1–8. IEEE, 2013.

[4] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *International*

Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 319–336. Springer, 2017.

[5] Andrea Arcuri. Evolutionary repair of faulty software. *Applied Soft Computing*, 11(4):3494–3514, 2011.

[6] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[7] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *International Symposium on Foundations of Software Engineering*, pages 306–317. ACM, 2014.

[8] Earl T Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *International Symposium on Software Testing and Analysis*, pages 257–269. ACM, 2015.

[9] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, Cesare Tinelli, et al. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[10] Marcel Böhme and Abhik Roychoudhury. Corebench: Studying complexity of regression errors. In *International Symposium on Software Testing and Analysis*, pages 105–115. ACM, 2014.

[11] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Where is the bug and how is it fixed? an experiment with practitioners. In *Joint Meeting on Foundations of Software Engineering*, pages 117–128. ACM, 2017.

[12] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation*, volume 8, pages 209–224, 2008.

[13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[14] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C Rinard. Detecting and escaping infinite loops with jolt. In *European Conference on Object-Oriented Programming*, pages 609–633. Springer, 2011.

[15] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *International Conference on Software Engineering*, pages 121–130. IEEE, 2011.

[16] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *International Conference on Dependable Systems and Networks*, pages 595–604. IEEE, 2002.

[17] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In *International Con-*

ference on Tools and Algorithms for the Construction and Analysis of Systems, pages 93–107. Springer, 2013.

[18] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *International Conference on Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[19] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[20] Cristina David and Daniel Kroening. Program synthesis: challenges and opportunities. *Phil. Trans. R. Soc. A*, 375(2104):20150403, 2017.

[21] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[22] Brian Demsky, Michael D Ernst, Philip J Guo, Stephen McCamant, Jeff H Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *International symposium on Software testing and analysis*, pages 233–244. ACM, 2006.

[23] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[24] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[25] Loris DAntoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification*, pages 383–401. Springer, 2016.

[26] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: a tool for repairing complex data structures. In *International Conference on Software engineering*, pages 855–858. ACM, 2008.

[27] Evren Ermis, Martin Schäf, and Thomas Wies. Error invariants. In *International Symposium on Formal Methods*, pages 187–201. Springer, 2012.

[28] Michael Fagan. Design and code inspections to reduce errors in program development. In *Software pioneers*, pages 575–607. Springer, 2002.

[29] Joseph Feller, Brian Fitzgerald, et al. *Understanding open source software development.* Addison-Wesley London, 2002.

[30] Zachary P Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2012.

[31] Zhaohui Fu and Sharad Malik. On solving the partial max-sat problem. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 252–265. Springer, 2006.

[32] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. Fixing recurring crash bugs via analyzing q&a sites. In *International Conference on Automated Software Engineering*, pages 307–318. IEEE, 2015.

[33] Patrice Godefroid. Higher-order test generation. In *Conference on Programming Language Design and Implementation*. ACM, 2011.

[34] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using sat. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–188. Springer, 2011.

[35] Haifeng He and Neelam Gupta. Automated debugging using path-based weakest preconditions. In *International Conference on Fundamental Approaches to Software Engineering*, pages 267–280. Springer, 2004.

[36] Jinru Hua and Sarfraz Khurshid. A sketching-based approach for debugging using test cases. In *International Symposium on Automated Technology for Verification and Analysis*, pages 463–478. Springer, 2016.

[37] Joxan Jaffar, Jorge A Navas, and Andrew E Santosa. Unbounded symbolic execution for program verification. In *International Conference on Runtime Verification*, pages 396–411. Springer, 2011.

[38] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *International Conference on Software Engineering*, pages 215–224. ACM, 2010.

[39] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Conference on Programming Language Design and Implementation*. ACM, 2011.

[40] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *International Conference on Computer Aided Verification*, pages 226–238. Springer, 2005.

[41] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *International Conference on Automated software engineering*, pages 273–282. ACM, 2005.

[42] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *International conference on Software engineering*, pages 467–477. ACM, 2002.

[43] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. 2011.

[44] René Just, Michael D Ernst, and Gordon Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In

International Symposium on Software Testing and Analysis, pages 315–326. ACM, 2014.

[45] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. Minthint: Automated synthesis of repair hints. In *International Conference on Software Engineering*, pages 266–276. ACM, 2014.

[46] Ming Kawaguchi, Shuvendu K Lahiri, and Henrique Rebelo. Conditional equivalence. *MSR Tech. Rep*, 2010.

[47] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *International Conference on Automated Software Engineering*, pages 295–306. IEEE, 2015.

[48] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[49] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[50] Robert Könighofer and Roderick Bloem. Automated error localization and correction for imperative programs. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 91–100. FMCAD Inc, 2011.

[51] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative

programs. In *International Conference on Computer Aided Verification*, pages 712–717. Springer, 2012.

[52] Shuvendu K Lahiri, Rohit Sinha, and Chris Hawblitzel. Automatic rootcausing for program equivalence failures in binaries. In *International Conference on Computer Aided Verification*, pages 362–379. Springer, 2015.

[53] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Conference on Programming Language Design and Implementation*. ACM, 2014.

[54] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Joint Meeting on Foundations of Software Engineering*, pages 593–604. ACM, 2017.

[55] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *International Conference on Software Analysis, Evolution, and Reengineering*, volume 1, pages 213–224. IEEE, 2016.

[56] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, pages 3–13. IEEE, 2012.

[57] Claire Le Goues, Westley Weimer, and Stephanie Forrest. Representations and operators for improving evolutionary software repair. In

Conference on Genetic and Evolutionary Computation, pages 959–966. ACM, 2012.

[58] Rainer Leupers. *Code optimization techniques for embedded processors: Methods, algorithms, and tools.* Springer Science & Business Media, 2013.

[59] Qing Li, Tatuya Jinmei, and Keiichi Shima. *IPv6 Core Protocols Implementation.* Morgan Kaufmann, 2010.

[60] Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. Bug isolation via remote program sampling. In *Conference on Programming Language Design and Implementation.* ACM, 2003.

[61] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Joint Meeting on Foundations of Software Engineering*, pages 727–739. ACM, 2017.

[62] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[63] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. 2016.

[64] Paul Dan Marinescu and Cristian Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 716–726. IEEE Press, 2012.

[65] Yuri Matiyasevich. Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR*, 191:279–282, 1970.

[66] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. Symbolic execution with existential second-order constraints. In *Joint Meeting on Foundations of Software Engineering*. ACM, 2018.

[67] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. 2018.

[68] Sergey Mechtaev, Gao Xiang, Shin Hwei Tan, and Abhik Roychoudhury. Partitioning patches into test-equivalence classes for scaling program repair. *arXiv:1707.03139*, 2017.

[69] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *International Conference on Software Engineering*, pages 448–458. IEEE Press, 2015.

[70] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering*, pages 691–701. IEEE, 2016.

[71] Elliott Mendelson. *Introduction to mathematical logic.* CRC press, 2009.

[72] Martin Monperrus. A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair. In *International Conference on Software Engineering*, pages 234–242. ACM, 2014.

[73] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[74] ThanhVu Nguyen, Westley Weimer, Deepak Kapur, and Stephanie Forrest. Connecting program synthesis and reachability: Automatic program repair using test-input generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 301–318. Springer, 2017.

[75] Gene Novark, Emery D Berger, and Benjamin G Zorn. Exterminator: Automatically correcting memory errors with high probability. *Communications of the ACM*, 51(12):87–95, 2008.

[76] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. Shadow of a doubt: testing for divergences between software versions. In *International Conference on Software Engineering*, pages 1181–1192. ACM, 2016.

[77] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *International symposium on software testing and analysis*, pages 199–209. ACM, 2011.

[78] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically patching errors in deployed software. In *Symposium on Operating systems principles*, pages 87–102. ACM, 2009.

[79] Suzette Person, Matthew B Dwyer, Sebastian Elbaum, and Corina S Psreanu. Differential symbolic execution. In *International Symposium on Foundations of software engineering*, pages 226–237. ACM, 2008.

[80] Justyna Petke, Mark Harman, William B Langdon, and Westley Weimer. Using genetic improvement and code transplants to specialise a c++ program to a problem class. In *European Conference on Genetic Programming*, pages 137–149. Springer, 2014.

[81] Ranjith Purushothaman and Dewayne E Perry. Toward understanding the rhetoric of small source code changes. *Transactions on Software Engineering*, 31(6):511–526, 2005.

[82] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: An approach to debugging evolving programs. *Transactions on Software Engineering and Methodology*, 21(3):19, 2012.

[83] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[84] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis*, pages 191–201. ACM, 2013.

[85] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis*, pages 24–36. ACM, 2015.

[86] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. Counterexample-guided quantifier instantiation for synthesis in smt. In *International Conference on Computer Aided Verification*, pages 198–216. Springer, 2015.

[87] Hesam Samimi, Ei Darli Aung, and Todd Millstein. Falling back on executable specifications. In *European Conference on Object-Oriented Programming*, pages 552–576. Springer, 2010.

[88] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *International Conference on Software Engineering*, pages 277–287. IEEE, 2012.

[89] Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *Architectural Support for Programming Languages and Operating Systems*, pages 317–328. ACM, 2013.

[90] Stelios Sidiroglou and Angelos D Keromytis. Countering network worms through automatic patch generation. *Security & Privacy*, 3(6):41–49, 2005.

[91] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. Codecarboncopy. In *Joint Meeting on Foundations of Software Engineering*, pages 95–105. ACM, 2017.

[92] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Conference on Programming Language Design and Implementation*. ACM, 2015.

[93] João P Marques Silva and Karem A Sakallah. Graspa new search algorithm for satisfiability. In *International conference on Computer-aided design*, pages 220–227. IEEE Computer Society, 1997.

[94] Alexey Smirnov and Tzi-cker Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *The Network and Distributed System Security Symposium*, 2005.

[95] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015.

[96] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. 2006.

[97] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software testing foundations: a study guide for the certified tester exam.* Rocky Nook, Inc., 2014.

[98] Kathryn T Stolee, Sebastian Elbaum, and Daniel Dobos. Solving the search for source code. *Transactions on Software Engineering and Methodology*, 23(3):26, 2014.

[99] Kathryn T Stolee, Sebastian Elbaum, and Matthew B Dwyer. Code search with input/output queries: Generalizing, ranking, and assessment. *Journal of Systems and Software*, 116:35–48, 2016.

[100] Shin Hwei Tan and Abhik Roychoudhury. relifix: Automated repair of software regressions. In *International Conference on Software Engineering*, pages 471–482. IEEE Press, 2015.

[101] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *International Symposium on Foundations of Software Engineering*, pages 727–738. ACM, 2016.

[102] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[103] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. In *International conference on tests and proofs*, pages 134–153. Springer, 2008.

[104] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Joint Meeting on Foundations of Software Engineering*, pages 253–262. ACM, 2005.

[105] Gregory Tseytin. On the complexity of derivation in propositional calculus. *Studies in Constrained Mathematics and Mathematical Logic*, 1968.

[106] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *International symposium on Software testing and analysis*, pages 61–72. ACM, 2010.

[107] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *International Conference on Automated Software Engineering*, pages 356–366. IEEE, 2013.

[108] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

[109] Peter Weißgerber, Daniel Neu, and Stephan Diehl. Small patches get in! In *International working conference on Mining software repositories*, pages 67–76. ACM, 2008.

[110] Qi Xin and Steven P Reiss. Identifying test-suite-overfitted patches through test case generation. In *International Symposium on Software Testing and Analysis*, pages 226–236. ACM, 2017.

[111] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. 2018.

[112] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *International Conference on Software Engineering*, pages 416–426. IEEE Press, 2017.

[113] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *Transactions on Software Engineering*, 43(1):34–55, 2017.

[114] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Joint Meeting on Foundations of Software Engineering*, pages 831–841. ACM, 2017.

[115] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. In *Joint Meeting on Foundations of Software Engineering*, pages 740–751. ACM, 2017.

[116] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. A correlation study between automated program

repair and test-suite metrics. *Empirical Software Engineering*, pages 1–32, 2017.

[117] Tianyi Zhang and Miryung Kim. Automated transplantation and differential testing for clones. In *International Conference on Software Engineering*, pages 665–676. IEEE Press, 2017.