Automatic Program Repair Using Formal Verification and Expression Templates

Thanh-Toan Nguyen(✉), Quang-Trung Ta, and Wei-Ngan Chin

School of Computing, National University of Singapore, Singapore, Singapore
{toannt,taqt,chinwn}@comp.nus.edu.sg

Abstract. We present an automated approach to repair programs using formal verification and expression templates. In our approach, an input program is first verified against its formal specification to discover potentially buggy statements. For each of these statements, we identify the expression that needs to be repaired and set up a template patch which is a linear expression composed of the program's variables and unknown coefficients. Then, we analyze the template-patched program against the original specification to collect a set of constraints of the template patch. This constraint set will be solved by a constraint solving technique using Farkas' lemma to identify the unknown coefficients, consequently discovering the actual patch. We implement our approach in a tool called Maple and evaluate it with various buggy programs from a widely used benchmark TCAS, and a synthetic yet challenging benchmark containing recursive programs. Our tool can quickly discover the correct patches and outperforms the state-of-the-art program repair tools.

1 Introduction

The last decade has witnessed the rapid development of automatic program repair, an active research area in computer science [19]. The goal of this research field is to automatically generate patches to fix bugs in software programs. Researchers have applied a common approach which uses test suites to localize bugs, and then generate and validate patches. This test-suite-based method is used by many works, such as [11,14,15,17]. However, this approach might produce *overfitting patches*: fixes that can pass all test cases, but also might break untested yet desired functionality of programs. Therefore, the quality of output patches often depends on the coverage of the provided test suites [22].

To avoid the above limitation of the test-suite-based approach, other researchers proposed to leverage formal specification to guide the repair process. This approach is used in several works like [8,9,13,18,21,25]. In this method, the correctness of a program can be specified by logical formulas, which appear in forms of pre-conditions, post-conditions, assertions, and invariants. Then, a deductive verification system is deployed to check the input program against its provided specifications to localize bugs and generate patches. In comparison to the test-suite-based approach, the formal-specification-based method provides

C. Enea and R. Piskac (Eds.): VMCAI 2019, LNCS 11388, pp. 70–91, 2019.
https://doi.org/10.1007/978-3-030-11245-5_4

better coverage on the relation of the program's input/output. However, the current solutions to generate patches are still limited. For example, Rothenberg and Grumberg [21] use simple code mutations to generate patches while Kneuss et al. [8] need the tests of corner cases to repair functional programs.

In this work, we follow the formal-specification-based approach to repair faulty C programs. We propose a general solution to discover patches by using expression templates and constraint solving. Our method is summarized as follows. We first invoke a deductive system to verify an input program against its specification. If this program fails to meet its specification, we obtain a set of invalid proof obligations, which can be utilized to locate potentially buggy statements. Here, we consider the bug type related to arithmetic expression, which can be the test expression of a branching or a loop statement, or the expression in the right hand side of an assignment. We replace each possibly buggy expression by a template patch which is a linear expression of the program's variables and unknown coefficients to create a template program. This program will be analyzed against the original specification to collect a set of proof obligations containing the template patch. These proof obligations will be solved to determine the actual values of the unknown coefficients, thus discover the repaired program.

Contributions. Our work makes the following contributions.

- We propose an automatic framework to repair programs using formal specification and expression templates. The use of formal verification enables our framework to locate buggy statements faster and more precise than other testing-based approaches.
- We propose a novel method to generate program patches using expression templates and constraint solving. Our solution is more general than existing approaches that perform only simple code mutations.
- We implement the proposed approach in a tool, called Maple, and experiment with it on a widely used benchmark named TCAS and a challenging synthetic benchmark of recursive programs. Our tool can repair a majority of the programs in these benchmarks and outperforms the state-of-the-art program repair tools. Moreover, it does not introduce any overfitting patch.

2 Motivating Example

We consider a simple C program `sum` which computes the sum of all natural numbers from `0` to a given input number `n` (Fig. 2). This program is specified by a pair of pre-condition and post-condition, captured by keywords `requires`/`ensures` (lines 2, 3). In essence, this specification indicates that given a non-negative input `n`, or $\texttt{n} \geq 0$, the expected output, represented by the variable `res`, is $\texttt{n} \cdot (\texttt{n} + 1)/2$.

The body of `sum` is implemented in a recursive fashion (lines 4–10). In the base case, when the input `n` is `0`, this program returns `0` (line 5). Otherwise, in the recursive case, it first computes the sum `s` of all natural numbers from `0` to `n − 1` (line 7), and adds `2 ∗ n` to that sum (line 8). However, this implementation of the recursive case is *buggy*. In line 8, by adding `2 ∗ n` to `s`, the final result of the procedure `sum(n)` cannot be equal to `n · (n + 1)/2`, as specified in the post-condition (line 3).

```
1:  int sum(int n)
2:  //@ requires n ≥ 0
3:  //@ ensures res = n·(n + 1)/2
4:  {
5:     if (n == 0) return 0;
6:     else {
7:        int s = sum(n − 1);
8:        return 2 ∗ n + s;
9:     }
10: }
```

Fig. 1. A faulty C program

Given the specification in lines 2, 3, existing verification tools such as [1,12] can easily detect the bug at line 8. However, these tools do not support repairing faulty programs. Moreover, repairing this bug is challenging, and the state-of-the-art program repair tools cannot discover a patch that replaces `2 ∗ n` by `n`. There are two reasons as follows. Firstly, this patch cannot be discovered by the technique that performs simple code mutation [21], since it does not consider mutating the variables' coefficients. Even if the coefficient mutation is supported, it is still impractical to discover the correct patch since the number of possible values for these coefficients is infinite. Secondly, the program `sum` contains a recursive call, which is challenging for the test-suite-based methods [11,15]. For instance, genetic programming operators used by GenProg [11], such as deletion, swap, or insertion, suffer the same difficulty as the mutation-based counterpart in finding the correct coefficients.

We observe that the desired patch should be an expression of some variables in the program. In particular, it can be an expression of at most two variables `s` and `n`. Here, we focus on finding the patch in form of a *linear expression*. Therefore, we denote the desired patch by an expression $\mathtt{f(s,n)} \triangleq \mathtt{c}_1 * \mathtt{s} + \mathtt{c}_2 * \mathtt{n} + \mathtt{c}_3$, where $\mathtt{c}_1, \mathtt{c}_2, \mathtt{c}_3$ are some unknown integer coefficients.

```
1:  int sum(int n)
2:  //@ requires n ≥ 0
3:  //@ ensures res = n·(n + 1)/2
4:  {
5:     if (n == 0) return 0;
6:     else {
7:        int s = sum(n − 1);
8:        return f(s, n); // a template fix
9:     }
10: }
```

Fig. 2. A template fix for the program `sum`

Now, we can apply standard verification techniques [1,12] to collect the proof obligations about `f(s, n)`, which need to be valid so that the program satisfies its specification. These proof obligations will be solved to discover the actual values of the unknown coefficients $\mathtt{c}_1, \mathtt{c}_2, \mathtt{c}_3$. We will elaborate the details in Sect. 4.

3 Background

In this section, we represent the background of the formal verification of software. We target to the class of C programs that performs logical and arithmetic computations. The program syntax can be referred to in the C11 standard [6]. In our approach, the functional correctness of a program is represented by a specification, which are logical formulas preceded by the special string "//@", as shown in the motivating example in Fig. 2.

Our specification language is presented in Fig. 3. We write c, x to denote an integer constant, variable, and res is a special variable representing the output of a procedure. The expression e is constructed using basic arithmetic operations: addition, subtraction, multiplication, division. We write P to indicate a first-order logic formula, which is composed of equality and arithmetic constraints, using standard logical connectives and quantifications. Finally, $\mathcal{S}$ denotes a specification which is either a pair of pre-condition and post-condition of a procedure (preceded by the keywords `requires` and `ensures`) or an invariant of a loop statement (preceded by the keyword `invariant`).

$$\begin{aligned}
e &::= c \mid x \mid res \mid -e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \cdot e_2 \mid e_1 \,/\, e_2 \\
P &::= \mathsf{true} \mid \mathsf{false} \mid e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 > e_2 \mid e_1 \geq e_2 \mid e_1 < e_2 \mid e_1 \leq e_2 \\
&\quad\;\; \mid \neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \rightarrow P_2 \mid \forall x.P \mid \exists x.P \\
\mathcal{S} &::= \texttt{requires}\ P_1\ \texttt{ensures}\ P_2 \mid \texttt{invariant}\ P
\end{aligned}$$

Fig. 3. Syntax of the specification language

We follow the literature to use Hoare logic [4] to verify the functional correctness of a program against its specification. The heart of this logic is a Hoare triple $\{P\}$ `C` $\{Q\}$ which describes how a program changes its state during the execution. Here P and Q are two assertions, representing the pre-condition and post-condition of the program `C`. In essence, the Hoare triple $\{P\}$ `C` $\{Q\}$ states that for a given program state satisfying P, if the program `C` executes and terminates, then the new program state will satisfy Q.

Hoare logic provides inference rules for all the constructs of an imperative programming language. They include the rules handling assignment, sequential composition of statements, branching statements, function call, etc. These rules are standard and can be found in many works in the field of program verification, such as [4,5]. For example, the rule for the composition of statements and the `if` statement are presented in Fig. 4. Interested readers can refer to [5] for more Hoare rules.

$$\frac{\{P\}\ \mathtt{C_1}\ \{Q\} \qquad \{Q\}\ \mathtt{C_2}\ \{R\}}{\{P\}\ \ \mathtt{C_1;C_2}\ \ \{R\}}\ \mathtt{composition} \qquad \frac{\{P \wedge R\}\ \mathtt{C_1}\ \{Q\} \qquad \{P \wedge \neg R\}\ \mathtt{C_2}\ \{Q\}}{\{P\}\ \ \mathtt{if}\ (R)\ \mathtt{C_1}\ \mathtt{else}\ \mathtt{C_2}\ \ \{Q\}}\ \mathtt{if}$$

Fig. 4. Examples of Hoare rules

4 Our Approach to Repair Faulty Programs

We now elaborate our program repair approach. The workflow is illustrated in Fig. 5. Given a program and its specification, we verify the program symbolically, using Hoare logic, to determine whether it behaves correctly w.r.t. its specification. If the verification step fails, we localize the possibly buggy statements and create possible template patches, which are linear expressions of the program's variables with unknown coefficients. Each of the template-patched programs will be verified again to collect a set of constraints (proof obligations) over the corresponding template patch. Then, this constraint set will be solved by a constraint solving technique using Farkas' lemma to discover the unknown coeffients of the template patch. If a solution of these coeffients can be found, then a candidate patch to repair the program is obtained. This candidate will be validated against the specification to determine if it is the actual patch. If the selected buggy statement cannot be repaired, then the next possibly buggy statement will be examined.

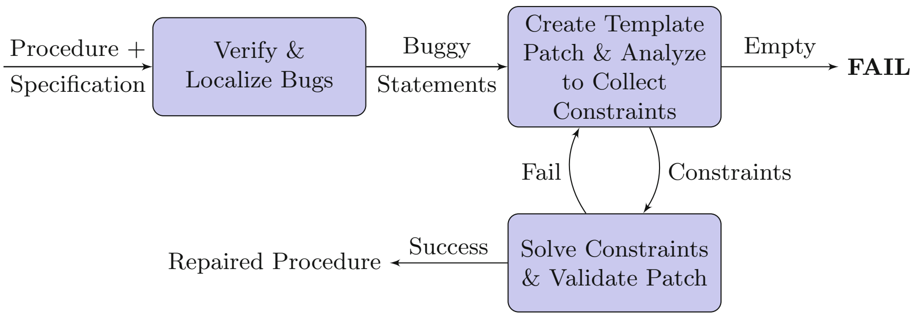

Fig. 5. Overview of Repair Procedure

In the next three Sects. 4.1, 4.2 and 4.3, we will describe the main components of our framework that verify and localize bug, create template patch and collect constraints, and solve constraints. Then, we will summarize our approach using a pseudo code algorithm in Sect. 4.4 and discuss its soundness in Sect. 4.5.

4.1 Verifying Programs and Localizing Bugs

We follow the literature to apply Hoare logic to verify programs and localize bugs. This approach is well-known and has been described in many works, such as [5]. We briefly summarize it as follows.

Program Verification. When symbolically analyzing a procedure of a program against its specification, we first assign its pre-condition to the initial program state of that procedure. Then, the program state after executing each statement of the procedure will be computed using the Hoare rules. Note that for each function call, the callee will not be analyzed directly. Instead, its specification will verified and utilized to update the program state of the caller.

For each loop statement of the procedure, we need to check if the program states at the entry and the exit of the loop imply the loop invariant. Similarly, for each `return` statement, we also need to check if the program state at the returned point implies the post-condition. The procedure is said to be *correct* w.r.t. its specification if all the aforementioned implications (proof obligations) can be proved valid. Otherwise, it is considered *buggy*.

For example, the verification of the program `sum` in Sect. 2 is presented in Fig. 6. We indicate the program states after executing each statement by the string "//". At the beginning, the initial program state is updated with the given pre-condition $n \geq 0$ (line 5). Then, the program executes the first branch of the `if` statement and the branching condition $n = 0$ is propagated (line 7).

```
 1: int sum(int n)
 2: //@ requires n≥0
 3: //@ ensures res = n·(n + 1)/2
 4: {
 5:   // n≥0                      (the initial program state is from the pre-condition)
 6:   if (n == 0)
 7:     // n≥0 ∧ n=0
 8:     return 0;
 9:     // n≥0 ∧ n=0 ∧ res=0                                    (the final program state)
10:     //                                       ⇒ need to prove the post-condition:
11:     //                                   n≥0 ∧ n=0 ∧ res=0 ⊢ res=n·(n+1)/2
12:   else {
13:     // n≥0 ∧ n≠0
14:     int s = sum(n − 1);
15:     // n≥0 ∧ n≠0 ∧ s=(n−1)·n/2                      (use the post-condition of sum)
16:     return 2 ∗ n + s;
17:     // n≥0 ∧ n≠0 ∧ s=(n−1)·n/2 ∧ res=2·n+s              (the final program state)
18:     //                                       ⇒ need to prove the post-condition:
19:     //              n≥0 ∧ n≠0 ∧ s=(n−1)·n/2 ∧ res=2·n+s ⊢ res=n·(n+1)/2
20:   }
21: }
```

Fig. 6. Verifying the motivating example

When the program exits (line 8), the constraint of the returned result $res = 0$ is accumulated to obtain the final program state $n \geq 0 \wedge n = 0 \wedge res = 0$. Now, the verification system needs to check if this program state implies the post-condition, that is, to prove the following entailment E_1:

$$E_1 \triangleq n \geq 0 \wedge n = 0 \wedge res = 0 \vdash res = n \cdot (n+1)/2$$

In our approach, the entailment E_1 can be easily proved by invoking an off-the-shelf SMT solver like Z3 [16]. In fact, this entailment is *valid* since the constraint $n = 0 \wedge res = 0$ in its antecedent implies the constraint $res = n \cdot (n+1)/2$ in its consequent. Hence, this execution path of the `if` statement is considered correct w.r.t. the specification.

Similarly, when the program executes the `else` branch (line 12), the branching condition $n \neq 0$ is also propagated to the program state (line 13). When the recursive call `sum(n - 1)` is performed (line 14), the verification system checks if the current program state $n \geq 0 \wedge n \neq 0$ implies the pre-condition $n - 1 \geq 0$ of this function call. After that, the post-condition $s = (n-1) \cdot n/2$ of this call will be accumulated into the current program state (line 15). When the program exits (line 16), the final program state is $n \geq 0 \wedge n \neq 0 \wedge s = (n-1) \cdot n/2 \wedge res = 2 \cdot n + s$. Again, the verification system also needs to check whether this program state implies the post-condition, resulting in the following entailment:

$$E_2 \triangleq n \geq 0 \wedge n \neq 0 \wedge s = (n-1) \cdot n/2 \wedge res = 2 \cdot n + s \vdash res = n \cdot (n+1)/2$$

However, this entailment is *invalid*, since its antecedent, which can be simplified to $n \geq 0 \wedge res = n \cdot (n+3)/2$, cannot prove its consequent $res = n \cdot (n+1)/2$. Consequently, there is a bug in this execution path of the program `sum`.

Bug Localization. Once the program is verified, we identify the invalid proof obligation to discover the buggy execution path. For example, the invalid proof obligation for the program `sum` (Fig. 6) is the entailment E_2 above. Thus, there is a bug in the execution path of the `else` branch.

In our implementation, we record the correspondence of the constraints in each proof obligation with the program specification and code. This record enables us to identify that the constraint $n \geq 0$ comes from the pre-condition (line 2, Fig. 6), $n \neq 0$ is from the `if` statement (line 6), etc. Using this record, we can simplify the antecedent of the invalid proof obligation by removing all constraints belonging to the program specification. The remaining constraints which correspond to the program code are the ones that cause the bug. For example, when removing the constraint $n \geq 0$ from E_2, we obtain the constraint F, which corresponds to all possible bugs of the program `sum`.

$$F \triangleq n \neq 0 \wedge s = (n-1) \cdot n/2 \wedge res = 2 \cdot n + s$$

To make the bug localization more efficient, we rank the remaining constraints by their likelihood to trigger the bug. Our ranking heuristics are as follows.

– If a constraint has its corresponding program code which belongs to a correct execution path, then this constraint is less likely to cause the bug in other execution paths.

– If a constraint has the corresponding program code belonging to only the buggy execution path, then this constraint is more likely to cause the bug.

For example, the constraint $n \neq 0$ in F corresponds to the conditional statement `if (n == 0)` (line 6), which also belong the correct execution path related to the proof obligation E_1. Therefore, the likelihood to cause the bug of $n \neq 0$ is low.

The two constraints $s = (n-1) \cdot n/2$ and $res = 2 \cdot n + s$ correspond to the function call `sum(n − 1)` (line 14) and the computation `2 * n + s` (line 16). Since these two statements appear only in the execution path related to the invalid proof obligation E_2, their likelihoods to cause the bug are equally high.

4.2 Creating Template Patches and Analyzing Template Programs

For each possibly buggy expression discovered in the previous step, we substitute it by a linear template patch to create a template-patched program. In essence, a patch is a linear expressions of the program variables in the execution path leading to the bug with unknown coefficients (Definition 1). Then, each template-patched program will be verified against its specification to obtain a set of entailments (proof obligations) related to the expression template.

Definition 1 (Linear Expression Template). *A linear expression template for n variables $x_1, ..., x_n$, denoted as $f(x_1, ..., x_n)$, is an expression of the form $c_1 \cdot x_1 + ... + c_n \cdot x_n + c_{n+1}$, where $c_1, ..., c_n, c_{n+1}$ are unknown integer coefficients.*

For example, given the possibly buggy expression `sum(n − 1)` discovered in the previous section (line 14, Fig. 6), there exists only 1 variable `n` in the execution path leading to the function call `sum(n − 1)`. Therefore, we can create a linear expression template $f(n) \triangleq c_1 \cdot n + c_2$, which replaces the expression $n - 1$ to create a patch `sum(f(n))`.

Similarly, given the possibly buggy expression `2 * n + s` (line 16, Fig. 6), there exist two variables `s, n` involved in the corresponding execution path. Hence, we can create a template patch $f(s, n) \triangleq c_1 \cdot s + c_2 \cdot n + c_3$. We illustrate the template-patched program for this bug in Fig. 7.

After analyzing this program against its specification, we obtain a proof obligation set containing one entailment: $n \geq 0 \wedge n \neq 0 \wedge s = (n-1) \cdot n/2 \wedge res = f(s, n) \vdash res = n \cdot (n+1)/2$. This entailment can be rewritten as the entailment E_3 below by unfolding the definition of the expression template $f(s, n) \triangleq c_1 \cdot s + c_2 \cdot n + c_3$:

$$\begin{aligned} E_3 \triangleq\ & n \geq 0 \wedge n \neq 0 \wedge s = (n-1) \cdot n/2 \\ & \wedge res = c_1 \cdot s + c_2 \cdot n + c_3 \vdash res = n \cdot (n+1)/2 \end{aligned}$$

```
int sum(int n)
//@ requires n≥0
//@ ensures res = n·(n + 1)/2
{
  // n≥0                                                    (the initial program state)
  if (n == 0) return 0;
  else {
    // n≥0 ∧ n≠0
    int s = sum(n − 1);
    // n≥0 ∧ n≠0 ∧ s=(n−1)·n/2
    return f(s, n);                                                 // a template patch
    // n≥0 ∧ n≠0 ∧ s=(n−1)·n/2 ∧ res=f(s, n)                  (the final program state)
    //                                           ⇒ need to prove the post-condition:
    //              n≥0 ∧ n≠0 ∧ s=(n−1)·n/2 ∧ res=f(s, n) ⊢ res=n·(n+1)/2
  }
}
```

Fig. 7. Verifying the template-patched program

4.3 Solving Constraints to Discover Repaired Programs

In this section, we will describe the underlying constraint solving technique using Farkas' lemma [2]. We first restate Farkas' lemma and then explain how it is applied to solve a set $\mathcal{E}$ of entailments (proof obligations) collected from the verification of template-patched programs.

Theorem 1 (Farkas' Lemma). *Given a system S of linear constraints over real-valued variables $x_1, ..., x_n$:*

$$S \triangleq \bigwedge_{j=1}^{m} \sum_{i=1}^{n} a_{ij} \cdot x_i + b_j \geq 0.$$

When S is satisfiable, it entails the following linear constraint ψ:

$$\psi \triangleq \sum_{i=1}^{n} c_i \cdot x_i + \gamma \geq 0$$

if and only if there exists non-negative numbers $\lambda_1, ..., \lambda_m$ such that

$$\bigwedge_{i=1}^{n} c_i = \sum_{j=1}^{m} \lambda_j \cdot a_{ij} \quad \textit{and} \quad \sum_{j=1}^{m} \lambda_j \cdot b_j \leq \gamma$$

Given the set $\mathcal{E}$ of entailments, which contain unknown coefficients of the template patch, we can solve it in three steps:

- Normalize the entailments in $\mathcal{E}$ into entailments of the form $S \vdash \psi$, which satisfies the conditions of Farkas' lemma, where S is a conjunction of linear constraints, and ψ is a linear constraint.

– Apply Farkas' lemma to eliminate universal quantification to obtain new constraints with only existential quantification over the unknown coefficients and the factors λ_j.
– Solve the new constraints by an off-the-shelf prover, such as Z3 [16], to find the concrete values of the unknown coefficients in the template patch.

We now illustrate these 3 steps with the entailment E_3 collected in Sect. 4.2 to discover the unknown coefficients c_1, c_2, and c_3 of the expression template $f(s, n) \triangleq c_1 \cdot s + c_2 \cdot n + c_3$.

4.3.1 Normalizing the Entailments

In our work, the entailments obtained from the verification process might contain polynomial terms and equality/inequality relations. Therefore, we need to normalize them into the linear constraint forms satisfying the condition of Farkas' lemma. This normalization includes four steps: (1) linearizing all non-linear expressions, (2) transforming all arithmetic constraints to the form $e \geq 0$, (3) eliminating disjunctions in the antecedent of each entailment, and (4) transforming the consequent of each entailment to contain only one linear constraint. They are explained follows.

1. Linearizing non-linear expressions. We use the associative and distributive properties of arithmetic to unfold and simplify all non-linear expressions into polynomials. Then, we encode each polynomial term whose degree is greater than 1 ($k \cdot x_1^{k_1} \cdot \ldots \cdot x_n^{k_n}$ where $k_1 + \ldots + k_n > 1$) by an expression of its coefficient and a fresh variable ($k \cdot x'$, where x' is a fresh variable). For example, by applying this linearization, we can transform the entailment E_3 into the following entailment E_3', where u is a fresh variable that encodes n^2:

$$\begin{aligned} E_3' \triangleq\ & n \geq 0 \wedge n \neq 0 \wedge s = \tfrac{1}{2} \cdot u - \tfrac{1}{2} \cdot n \\ & \wedge res = c_1 \cdot s + c_2 \cdot n + c_3 \vdash res = \tfrac{1}{2} \cdot u + \tfrac{1}{2} \cdot n \end{aligned}$$

Note that in the linearization above, we do not capture the constraint between the new and the old variables. Hence, once the normalized entailments are solved to discover the unknown coefficients, we will need to validate if the discovered coefficients is also the solution of the original entailments. This detail will be discussed again in Sect. 4.3.3.

2. Transforming arithmetic constraints. We can apply the following equivalence transformations of arithmetic constraints (over integer domain) to obtain the constraints of the form $e \geq 0$, which are required by Farkas' lemma.

$$\begin{array}{ll} e_1 = e_2 \equiv (e_1 - e_2 \geq 0) \wedge (e_2 - e_1 \geq 0) & e_1 \geq e_2 \equiv e_1 - e_2 \geq 0 \\ e_1 \neq e_2 \equiv (e_1 - e_2 - 1 \geq 0) \vee (e_2 - e_1 - 1 \geq 0) & e_1 < e_2 \equiv e_2 - e_1 - 1 \geq 0 \\ e_1 > e_2 \equiv e_1 - e_2 - 1 \geq 0 & e_1 \leq e_2 \equiv e_2 - e_1 \geq 0 \end{array}$$

By applying the above equivalences, we can transform the entailment E_3' into the following entailment E_3'':

$$E_3'' \triangleq n \geq 0 \wedge (n-1 \geq 0 \vee -n-1 \geq 0) \wedge s - \tfrac{1}{2} \cdot u + \tfrac{1}{2} \cdot n \geq 0 \wedge \tfrac{1}{2} \cdot u - \tfrac{1}{2} \cdot n - s \geq 0$$
$$\wedge\; res - c_1 \cdot s - c_2 \cdot n - c_3 \geq 0 \wedge c_1 \cdot s + c_2 \cdot n + c_3 - res \geq 0$$
$$\vdash\; res - \tfrac{1}{2} \cdot u - \tfrac{1}{2} \cdot n \geq 0 \wedge \tfrac{1}{2} \cdot u + \tfrac{1}{2} \cdot n - res \geq 0$$

3. Eliminating disjunctions in the entailments' antecedents. The disjunction operators in the antecedent of each entailment can be easily eliminated to introduce simpler entailments. In particular, we can replace an entailment like $F_1 \vee F_2 \vdash F_3$ in the entailment set $\mathcal{E}$ by two new entailments $F_1 \vdash F_3$ and $F_2 \vdash F_3$. This disjunction elimination preserves the validity of $\mathcal{E}$, since the entailment $F_1 \vee F_2 \vdash F_3$ is valid if and only if both $F_1 \vdash F_3$ and $F_2 \vdash F_3$ are valid. For example, by applying this transformation to E_3'', we obtain the set of two entailments below:

$$E_{31}'' \triangleq n \geq 0 \wedge n-1 \geq 0 \wedge s - \tfrac{1}{2} \cdot u + \tfrac{1}{2} \cdot n \geq 0 \wedge \tfrac{1}{2} \cdot u - \tfrac{1}{2} \cdot n - s \geq 0$$
$$\wedge\; res - c_1 \cdot s - c_2 \cdot n - c_3 \geq 0 \wedge\; c_1 \cdot s + c_2 \cdot n + c_3 - res \geq 0$$
$$\vdash res - \tfrac{1}{2} \cdot u - \tfrac{1}{2} \cdot n \geq 0 \wedge \tfrac{1}{2} \cdot u + \tfrac{1}{2} \cdot n - res \geq 0$$

$$E_{32}'' \triangleq n \geq 0 \wedge -n-1 \geq 0 \wedge s - \tfrac{1}{2} \cdot u + \tfrac{1}{2} \cdot n \geq 0 \wedge \tfrac{1}{2} \cdot u - \tfrac{1}{2} \cdot n - s \geq 0$$
$$\wedge\; res - c_1 \cdot s - c_2 \cdot n - c_3 \geq 0 \wedge\; c_1 \cdot s + c_2 \cdot n + c_3 - res \geq 0$$
$$\vdash res - \tfrac{1}{2} \cdot u - \tfrac{1}{2} \cdot n \geq 0 \wedge \tfrac{1}{2} \cdot u + \tfrac{1}{2} \cdot n - res \geq 0$$

4. Normalize the entailments' consequents. In this final step, we transform all entailments in $\mathcal{E}$ to the form whose consequents contain only 1 linear constraint. This can be done by applying the following transformation rules:

- If the entailment set $\mathcal{E}$ contains an entailment like $F_1 \vdash F_2 \wedge F_3$, then this entailment can be replaced by two new entailments $F_1 \vdash F_2$ and $F_1 \vdash F_3$.
- If $\mathcal{E}$ contains an entailment like $F_1 \vdash F_2 \vee F_3$, then this entailment can be replaced by either $F_1 \vdash F_2$ or $F_1 \vdash F_3$. Here, we derive two new entailment sets $\mathcal{E}_1$ and $\mathcal{E}_2$ which respectively contain $F_1 \vdash F_2$ and $F_1 \vdash F_3$. These two sets $\mathcal{E}_1$ and $\mathcal{E}_2$ will be solved independently, and if one of them has a solution, this solution is also the solution of the original set $\mathcal{E}$.

For example, in the entailment set containing E_{31}'' and E_{32}'', the entailments' consequents have only the conjunction operator ($\wedge$). Hence, we can apply the above transformation rules to derive a set of the following 4 entailments.

$$E_{311}'' \triangleq n \geq 0 \wedge n-1 \geq 0 \wedge s - \tfrac{1}{2} \cdot u + \tfrac{1}{2} \cdot n \geq 0 \wedge \tfrac{1}{2} \cdot u - \tfrac{1}{2} \cdot n - s \geq 0$$
$$\wedge\; res - c_1 \cdot s - c_2 \cdot n - c_3 \geq 0 \wedge c_1 \cdot s + c_2 \cdot n + c_3 - res \geq 0$$
$$\vdash res - \tfrac{1}{2} \cdot u - \tfrac{1}{2} \cdot n \geq 0$$

$$
\begin{aligned}
E''_{312} \triangleq\ & n \geq 0 \wedge n - 1 \geq 0 \wedge s - \tfrac{1}{2} \cdot u + \tfrac{1}{2} \cdot n \geq 0 \wedge \tfrac{1}{2} \cdot u - \tfrac{1}{2} \cdot n - s \geq 0 \\
& \wedge res - c_1 \cdot s - c_2 \cdot n - c_3 \geq 0 \wedge c_1 \cdot s + c_2 \cdot n + c_3 - res \geq 0 \\
& \vdash \tfrac{1}{2} \cdot u + \tfrac{1}{2} \cdot n - res \geq 0 \\
E''_{321} \triangleq\ & n \geq 0 \wedge -n - 1 \geq 0 \wedge s - \tfrac{1}{2} \cdot u + \tfrac{1}{2} \cdot n \geq 0 \wedge \tfrac{1}{2} \cdot u - \tfrac{1}{2} \cdot n - s \geq 0 \\
& \wedge res - c_1 \cdot s - c_2 \cdot n - c_3 \geq 0 \wedge c_1 \cdot s + c_2 \cdot n + c_3 - res \geq 0 \\
& \vdash res - \tfrac{1}{2} \cdot u - \tfrac{1}{2} \cdot n \geq 0 \\
E''_{322} \triangleq\ & n \geq 0 \wedge -n - 1 \geq 0 \wedge s - \tfrac{1}{2} \cdot u + \tfrac{1}{2} \cdot n \geq 0 \wedge \tfrac{1}{2} \cdot u - \tfrac{1}{2} \cdot n - s \geq 0 \\
& \wedge res - c_1 \cdot s - c_2 \cdot n - c_3 \geq 0 \wedge c_1 \cdot s + c_2 \cdot n + c_3 - res \geq 0 \\
& \vdash \tfrac{1}{2} \cdot u + \tfrac{1}{2} \cdot n - res \geq 0
\end{aligned}
$$

4.3.2 Generating the New Constraints

Once all the entailments are normalized into the form satisfying the conditions of Farkas' lemma (Theorem 1), we can apply the lemma to eliminate the universal quantification over all variables, and generate the constraints containing only unknown coefficients and factors λ_i. Details about this constraint generation can be referred to in [2]. For instance, given the four entailments above, we can generate the following constraints of the unknown coefficient c_1, c_2, c_3 and the factors λ_i.

$$
\begin{aligned}
F_1 \triangleq\ & (-\lambda_3 \cdot c_1 + \lambda_4 \cdot c_1 + \lambda_5 - \lambda_6 = 0) \\
& \wedge (\lambda_1 + \lambda_2 - \lambda_3 \cdot c_2 + \lambda_4 \cdot c_2 + \tfrac{1}{2} \cdot \lambda_5 - \tfrac{1}{2} \cdot \lambda_6 = -\tfrac{1}{2}) \\
& \wedge (\lambda_3 - \lambda_4 = 1) \wedge (-\tfrac{1}{2} \cdot \lambda_5 + \tfrac{1}{2} \cdot \lambda_6 = -\tfrac{1}{2}) \\
& \wedge (-\lambda_2 - \lambda_3 \cdot c_3 + \lambda_4 \cdot c_3 \leq 0) \\
F_2 \triangleq\ & (-\lambda_9 \cdot c_1 + \lambda_{10} \cdot c_1 + \lambda_{11} - \lambda_{12} = 0) \\
& \wedge (\lambda_7 + \lambda_8 - \lambda_9 \cdot c_2 + \lambda_{10} \cdot c_2 + \tfrac{1}{2} \cdot \lambda_{11} - \tfrac{1}{2} \cdot \lambda_{12} = \tfrac{1}{2}) \\
& \wedge (\lambda_9 - \lambda_{10} = -1) \wedge (-\tfrac{1}{2} \cdot \lambda_{11} + \tfrac{1}{2} \cdot \lambda_{12} = \tfrac{1}{2}) \\
& \wedge (-\lambda_8 - \lambda_9 \cdot c_3 + \lambda_{10} \cdot c_3 \leq 0) \\
F_3 \triangleq\ & (-\lambda_{15} \cdot c_1 + \lambda_{16} \cdot c_1 + \lambda_{17} - \lambda_{18} = 0) \\
& \wedge (\lambda_{13} - \lambda_{14} - \lambda_{15} \cdot c_2 + \lambda_{16} \cdot c_2 + \tfrac{1}{2} \cdot \lambda_{17} - \tfrac{1}{2} \cdot \lambda_{18} = -\tfrac{1}{2}) \\
& \wedge (\lambda_{15} - \lambda_{16} = 1) \wedge (-\tfrac{1}{2} \cdot \lambda_{17} + \tfrac{1}{2} \cdot \lambda_{18} = -\tfrac{1}{2}) \\
& \wedge (-\lambda_{14} - \lambda_{15} \cdot c_3 + \lambda_{16} \cdot c_3 \leq 0) \\
F_4 \triangleq\ & (-\lambda_{21} \cdot c_1 + \lambda_{22} \cdot c_1 + \lambda_{23} - \lambda_{24} = 0) \\
& \wedge (\lambda_{19} - \lambda_{20} - \lambda_{21} \cdot c_2 + \lambda_{22} \cdot c_2 + \tfrac{1}{2} \cdot \lambda_{23} - \tfrac{1}{2} \cdot \lambda_{24} = \tfrac{1}{2}) \\
& \wedge (\lambda_{21} - \lambda_{22} = -1) \wedge (-\tfrac{1}{2} \cdot \lambda_{23} + \tfrac{1}{2} \cdot \lambda_{24} = \tfrac{1}{2}) \\
& \wedge (-\lambda_{20} - \lambda_{21} \cdot c_3 + \lambda_{22} \cdot c_3 \leq 0)
\end{aligned}
$$

4.3.3 Solving the New Constraints

The new constraints obtained from previous steps can be solved by a SMT solver, such as Z3 [16], to discover the actual values of the unknown coefficients. For instance, when solving the aforementioned constraints, we obtain a solution forthe unknown coefficients c_1, c_2, c_3 that $c_1 = 1$, $c_2 = 1$, $c_3 = 0$. When replacing these values to the template patch $\mathtt{f(s,n)} \triangleq \mathtt{c_1} \cdot \mathtt{s} + \mathtt{c_2} \cdot \mathtt{n} + \mathtt{c_3}$ in E_3, we obtain the following new entailment:

$$\bar{E}_3 \triangleq n \geq 0 \land n \neq 0 \land s = (n-1) \cdot n/2 \land res = s + n \vdash res = n \cdot (n+1)/2$$

Recall that during the linearization of non-linear expressions (Sect. 4.3.1), all polynomial terms are encoded by fresh variables. Since this encoding does not maintain the relations of the old and the new variables, we need to validate if the discovered solution obtained here still satisfies the original entailments. This validation can be easily done by invoking an SMT solver to prove the new entailments (like $\bar{E}_3$).

4.4 The Repair Algorithm

Figure 8 presents our main procedure $\mathsf{Repair}(\mathcal{P}, \mathcal{S})$. Its inputs include a buggy procedure $\mathcal{P}$ and a correct specification $\mathcal{S}$. There are three possible outputs as follows. Firstly, if $\mathcal{P}$ is correct w.r.t. its specification $\mathcal{S}$, then it does not need to be repaired, and the procedure simply returns `NONE`. Secondly, if $\mathcal{P}$ is buggy and can be repaired, then the procedure returns $\mathtt{PATCH}\langle\bar{\mathcal{P}}\rangle$ to indicate that $\bar{\mathcal{P}}$ is the repaired solution. Finally, the procedure returns `FAIL` if it cannot repair the buggy procedure $\mathcal{P}$.

The procedure Repair first verifies the input program $\mathcal{P}$ against its specification $\mathcal{S}$ by invoking an auxiliary procedure Verify (line 1). If the verification fails, then there exists a bug in the implementation of $\mathcal{P}$ w.r.t. its specification $\mathcal{S}$. Then, Repair will utilize the invalid proof obligation to discover all possibly

Procedure $\mathsf{Repair}(\mathcal{P}, \mathcal{S})$

Input: A procedure $\mathcal{P}$, and its correct specification $\mathcal{S}$.
Output: `NONE` if $\mathcal{P}$ is correct w.r.t. $\mathcal{S}$, $\mathtt{PATCH}\langle\bar{\mathcal{P}}\rangle$ if $\mathcal{P}$ is buggy and $\bar{\mathcal{P}}$ is the repaired solution, or `FAIL` if $\mathcal{P}$ is buggy but cannot be repaired.

1: **if** $\mathsf{Verify}(\mathcal{P}, \mathcal{S}) = \mathtt{FAIL}$ **then** *//$\mathcal{P}$ is buggy w.r.t. to its specs $\mathcal{S}$*
2: $\quad X \leftarrow \mathsf{GetInvalidProofObligation}(\mathcal{P}, \mathcal{S})$
3: $\quad \mathcal{E} \leftarrow \mathsf{LocalizeBuggyExpressions}(\mathcal{P}, X)$ *//all possible buggy exps*
4: $\quad$ **for** E **in** $\mathcal{E}$ **do** *//repair each buggy expression*
5: $\quad\quad T \leftarrow \mathsf{CreateTemplatePatch}(\mathcal{P}, E)$
6: $\quad\quad \mathcal{P}' \leftarrow \mathsf{CreateTemplateProgram}(\mathcal{P}, T)$
7: $\quad\quad \mathcal{C} \leftarrow \mathsf{VerifyAndCollectProofObligations}(\mathcal{P}', \mathcal{S})$
8: $\quad\quad$ **if** $\mathsf{HasSolution}(\mathcal{C}, T)$ **then**
9: $\quad\quad\quad \bar{T} \leftarrow \mathsf{GetSolution}(\mathcal{C}, T)$
10: $\quad\quad\quad \bar{\mathcal{P}} \leftarrow \mathsf{CreateRepairedProgram}(\mathcal{P}', \bar{T})$
11: $\quad\quad\quad$ **if** $\mathsf{Verify}(\bar{\mathcal{P}}, \mathcal{S}) = \mathtt{SUCCESS}$ **then**
12: $\quad\quad\quad\quad$ **return** $\mathtt{PATCH}\langle\bar{\mathcal{P}}\rangle$ *//discover a patch*
13: $\quad$ **return** `FAIL` *//cannot repair any buggy expression*
14: **else return** `NONE` *//$\mathcal{P}$ is correct w.r.t. to its specs $\mathcal{S}$, does not need to be repaired*

Fig. 8. The repair algorithm

buggy expressions (lines 2, 3). Then, it attempts to repair each of these expressions (lines 4–12).

For each possibly buggy expression E, the procedure Repair creates a template patch T (line 5), which is a linear expression of the program's variables and unknown coefficients, as described earlier in Sect. 4.2. This template patch T will replace E in the original program $\mathcal{P}$ to create a template program $\mathcal{P}$' (line 6). This template program will be verified again to collect a constraint set $\mathcal{C}$ of proof obligations about the template T. This constraint set will be solved by the technique using Farkas' lemmas (lines 8, 9). If a solution $\bar{T}$ of the template patch T is discovered, it will be used to create a repaired program $\bar{\mathcal{P}}$ (line 10). This repaired program will be validated against the specs $\mathcal{S}$ (line 11), and will be returned by the procedure Repair (line 12) if this validation succeeds.

On the other hand, the procedure Repair returns `FAIL` if it cannot repair any of the possibly buggy expressions (line 13). It also returns `NONE` if the original program $\mathcal{P}$ is correct w.r.t. the specification $\mathcal{S}$ (line 14).

4.5 Soundness

We claim that our program repair approach is sound. We formally state this soundness in the following Theorem 2.

Theorem 2 (Soundness). *Given a buggy program $\mathcal{P}$ and a specification $\mathcal{S}$, if the procedure* Repair *returns a program $\bar{\mathcal{P}}$, then this repaired program satisfies the specification $\mathcal{S}$.*

Proof. In our repair algorithm (Fig. 8), after solving the constraints to discover a candidate program (lines 8–10), we always verify this candidate against its specification $\mathcal{S}$ (line 11). Consequently, if the procedure Repair returns a repaired program $\bar{\mathcal{P}}$, this program always satisfies the specification $\mathcal{S}$.

5 Implementation and Experiment

We implement our program repair approach in a tool, called Maple, using the OCaml programming language. It is built on top of the verification system HIP [1] and the theorem prover Songbird [23,24]. We evaluate the performance of Maple on repairing faulty programs in a literature benchmark TCAS [3], which implements a traffic collision avoidance system for aircrafts. This benchmark is widely used in previous experiments of many program repair tools; it has a correct program of 142 lines of C code and 41 different faulty versions to simulate realistic bugs. However, the benchmark TCAS does not contain any loop or recursive call, a popular feature in modern programming languages. Therefore, we decide to compose a more challenging benchmark, called Recursion, which contain not only non-recursive but also recursive programs.

Our experiment was conducted on a computer with CPU Intel® Core™ i7-6700 (3.4 GHz), 8 GB RAM, and Ubuntu 16.04 LTS. We compare Maple against the state-of-the-art program repair tools for C programs, which are AllRepair

[21], Forensic [9,10], GenProg [11], and Angelix [15]. Among these tools, GenProg and Angelix rely on test suites, while AllRepair and Forensic use specifications (in the form of assertions) to repair programs. The details of our tool Maple and experiments are available online at https://maple-repair.github.io.

5.1 Experiment with the Benchmark TCAS

In this experiment, we evaluate all the tools with 41 faulty programs in the benchmark TCAS [3]. Since this benchmark was used before in the experiment of other tools AllRepair, Forensic, Angelix, and GenProg, we reuse their original settings in our experiment. Particularly, the specification-based tools AllRepair and Forensic keep the correct program along with the faulty versions to check the correctness of the repair candidates. On the other hand, the testing-based tools GenProg and Angelix use different test suites of 50 cases for each faulty program. For our tool Maple, we manually write the specification for the correct program and use this specification to repair the faulty versions.[1]

Table 1 presents the detailed results of our experiment. We report whether a tool can correctly repair a program (denoted by ✓), or repair the program by an overfitted patch (denoted by o)[2], or cannot repair it (denoted by –). We also record the runtime (in seconds) of each tool. Here, we do not set a timeout: a tool can run until either it returns a patch or informs that it fails to find any patch. In the summary rows, we report the total number of the correct and overfitting patches discovered by each tool, and the average time spent by each tool. The best result is highlighted in the **bold** typeface.

Our tool Maple can correctly repair 26/41 faulty programs and does not produce any overfitting patch. This is the *best* result among all participants. The tool AllRepair is the second best, which it can successfully repair 18 programs. Forensic and Angelix are the next best tools, and they can correctly repair 15 and 9 programs, respectively. Note that although Forensic and Angelix can repair in total 23 and 32 programs, respectively, many of them (8 and 23 programs) are repaired by overfitting patches. While these patches pass the test suites used by Angelix and Forensic, they change the desired behaviors of the original program. For instance, in the faulty program `v_2`, the tool Angelix incorrectly replaces the buggy expression `Up_Separation` + 300 by `Up_Separation` + 24, while the expected repaired expression is `Up_Separation` + 100.

Regarding the execution time, our tool Maple is the *second* fastest when it spends on average 155.3 s to repair a program. It is slower than AllRepair which spends averagely 16.9 s per program. Here, AllRepair uses a simple strategy to mutate operators and constants. In contrast, our tool needs to create a patch template, collect and solve the template's constraints to discover the actual patch. Nonetheless, these heavier computations enable Maple to correctly fix more programs than AllRepair. On

[1] Our specification contains 34 lines, while the original program has 142 lines of code.

[2] The correct and the overfitted patches are classified by comparing the similarity in the structures of the repaired and the originally correct programs.

Table 1. Experiment with the benchmark TCAS, where the participants are AllRepair(ARP), Angelix(AGL), GenProg(GPR), Forensic(FRS), and Maple(MPL)

Programs	**Repair Result**					**Repair Time (s)**				
	ARP	AGL	GPR	FRS	MPL	ARP	AGL	GPR	FRS	MPL
v_1	✓	✓	*o*	–	✓	1	46	800	–	104
v_2	–	*o*	–	✓	✓	–	114	–	28	98
v_3	✓	*o*	–	–	✓	1	131	–	–	224
v_4	–	✓	*o*	*o*	✓	–	11	445	51	139
v_5	–	✓	–	–	–	–	911	–	–	–
v_6	✓	✓	–	✓	✓	1	42	–	52	100
v_7	–	*o*	–	✓	✓	–	7938	–	43	100
v_8	–	*o*	–	✓	✓	–	27	–	36	105
v_9	✓	*o*	*o*	✓	✓	2	366	149	286	107
v_10	✓	*o*	*o*	✓	✓	4	737	487	770	107
v_11	–	*o*	–	–	✓	–	738	–	–	184
v_12	✓	*o*	–	–	✓	1	1079	–	–	1264
v_13	–	*o*	–	–	–	–	926	–	–	–
v_14	–	*o*	–	–	–	–	230	–	–	–
v_15	–	*o*	–	–	–	–	1718	–	–	–
v_16	✓	*o*	–	✓	✓	21	32	–	47	93
v_17	✓	–	–	✓	✓	38	–	–	43	96
v_18	–	–	–	✓	✓	–	–	–	52	97
v_19	–	–	*o*	✓	✓	–	–	258	35	99
v_20	✓	*o*	*o*	✓	✓	1	398	738	224	99
v_21	–	*o*	–	*o*	–	–	36	–	452	–
v_22	–	*o*	–	–	–	–	504	–	–	–
v_23	–	*o*	*o*	–	–	–	604	165	–	–
v_24	–	*o*	–	–	–	–	605	–	–	–
v_25	✓	*o*	*o*	✓	✓	1	37	120	364	111
v_26	–	✓	–	–	–	–	1098	–	–	–
v_27	–	✓	–	–	–	–	1179	–	–	–
v_28	✓	✓	–	✓	✓	67	338	–	180	101
v_29	–	–	–	–	✓	–	–	–	–	94
v_30	–	–	–	–	✓	–	–	–	–	98
v_31	✓	*o*	*o*	*o*	✓	1	15	171	491	84
v_32	✓	*o*	*o*	*o*	✓	1	26	62	544	99
v_33	–	–	–	–	–	–	–	–	–	–
v_34	–	*o*	–	*o*	–	–	260	–	1420	–
v_35	✓	✓	–	✓	✓	67	175	–	179	111
v_36	✓	–	–	*o*	–	90	–	–	1501	–
v_37	–	–	–	–	–	–	–	–	–	–
v_38	–	–	–	–	–	–	–	–	–	–
v_39	✓	*o*	*o*	✓	✓	1	218	184	367	111
v_40	✓	✓	–	*o*	✓	4	28	–	514	107
v_41	✓	*o*	–	*o*	✓	3	29	–	603	106
Correct (✓)	18	9	0	15	**26**	**16.9**	3615.0	–	180.4	155.3
Overfit (*o*)	**0**	23	11	8	**0**	–	729.0	325.4	697.0	–
Total (41)	18	**32**	11	23	26	**16.9**	1540.7	325.4	360.1	155.3

the other hand, the other tools Forensic, Angelix, and GenProg spend longer time to repair a program, compared to our tool Maple. These performances can be explained as follows. Firstly, Forensic also uses template patches, but its constraint solving technique requires an incremental counter-example-driven template refinement, which is less efficient than our approach of using Farkas' lemma. Secondly, Angelix utilizes a component-based repair synthesis algorithm, which is more costly than our method, in the context of repairing linear expressions. Finally, GenProg needs to heuristically mutate the original programs many times to find correct patches.

5.2 Experiment with the Benchmark Recursion

In this experiment, we evaluate all tools with the synthetic benchmark Recursion, which contains challenging arithmetic programs. This benchmark is presented in Table 2. We classify its programs into two categories: *non-recursive* and *recursive*. The *non-recursive* category includes programs that compute the maximum, the minimum, or the sum of two or three numbers, or the absolute value of a number. On the other hand, programs in the *recursive* category are constructed in a similar fashion to the motivating program `sum` (Sect. 2). They compute the sums of different sequences of numbers, which can be enumerated by an indexing number i, starting from 0 to a given number n. For example, they include a sequence of n consecutive numbers or a sequence of n products of the form $i \cdot (i+1)$. Although these recursive programs are relatively small (each program contains 5 to 9 lines of code), they are still challenging for the existing state-of-the-art program repair tools.

In order to evaluate GenProg and Angelix, we follow the tools' guidelines to create test suites of 10 cases. Note that these tools require the values of variables for every recursive call of the recursive programs. We also create specifications for the tool AllRepair and Forensic. They follow the same style of using assertions to compare the results of running the correct and buggy programs. For our tool Maple, we create a desired specification for each buggy program. These specifications are small, they contain only 2 lines per program.

Table 2 presents the experimental result with the benchmark Recursion. Our tool Maple can repair all 26 faulty programs and does not generate any overfitting patch. Furthermore, Maple outperforms the second and the third best tools Angelix and Forensic, which could correctly repair only 8 and 5 faulty programs, respectively. On the other hand, the two tools AllRepair and GenProg cannot repair any program. These tools perform only simple code mutations such as alternating Boolean or arithmetic operators, which are insufficient to handle these buggy programs. For the tool Forensic, although it exploits the correct programs to generate test suites to repair the faulty programs, these test suites cannot fully cover the underlying computations of these recursive programs. Consequently, Forensic can discover only overfitting patches, as shown with the *recursive* category in Table 2.

Regarding the runtime, Forensic is the fastest tool when it takes averagely 3.6 s to correctly repair a program. Our tool Maple is the second fastest which spends 5.6 s per correctly repaired program. Note that this average runtime also includes the time spent on recursive programs, which Forensic can produce only overfitting patches. Also, for every program that Forensic can repair correctly (`max_2_2`, `max_3_2`, `min_2_2`, `min_3_2`, `absolute_2`), our tool Maple spends less time than Forensic, thanks to the efficiency of the constraint solving technique

Table 2. Experiment with our numeric benchmark, where the participants are AllRepair(Arp), Angelix(Agl), GenProg(Gpr), Forensic(Frs), and Maple(Mpl)

	Programs	Repair Result					Repair Time (s)				
		Arp	Agl	Gpr	Frs	Mpl	Arp	Agl	Gpr	Frs	Mpl
non-recursive	max_2_1	–	–	–	*o*	✓	–	–	–	4	4
	max_2_2	–	✓	–	✓	✓	–	8	–	2	2
	max_3_1	–	–	–	–	✓	–	–	–	–	8
	max_3_2	–	✓	–	✓	✓	–	10	–	5	3
	min_2_1	–	–	–	–	✓	–	–	–	–	4
	min_2_2	–	✓	–	✓	✓	–	8	–	3	2
	min_3_1	–	–	–	–	✓	–	–	–	–	8
	min_3_2	–	✓	–	✓	✓	–	16	–	5	3
	sum_2_1	–	–	–	–	✓	–	–	–	–	2
	sum_2_2	–	–	–	–	✓	–	–	–	–	2
	sum_3_1	–	–	–	–	✓	–	–	–	–	2
	sum_3_2	–	–	–	–	✓	–	–	–	–	2
	absolute_1	–	✓	–	✓	✓	–	15	–	3	2
	absolute_2	–	–	–	–	✓	–	–	–	–	7
recursive	sum_n_1	–	✓	–	–	✓	–	38	–	–	4
	sum_n_2	–	–	–	*o*	✓	–	–	–	29	7
	sum_n_3	–	–	–	–	✓	–	–	–	–	7
	sum_n_4	–	–	–	*o*	✓	–	–	–	24	7
	conseq_1	–	✓	–	–	✓	–	35	–	–	4
	conseq_2	–	–	–	*o*	✓	–	–	–	28	11
	conseq_3	–	–	–	–	✓	–	–	–	–	11
	conseq_4	–	–	–	*o*	✓	–	–	–	23	11
	increment_1	–	✓	–	–	✓	–	51	–	–	4
	increment_2	–	–	–	–	✓	–	–	–	–	9
	increment_3	–	–	–	–	✓	–	–	–	–	10
	increment_4	–	–	–	–	✓	–	–	–	–	10
Correct	(✓)	0	8	0	5	**26**	–	22.6	–	**3.6**	5.6
Overfit	(*o*)	**0**	**0**	**0**	5	**0**	–	–	–	21.6	–
Total	(26)	0	8	0	10	**26**	–	22.6	–	12.6	5.6

using Farkas' lemma. In this benchmark, the runtime of all tools is smaller than that of the benchmark TCAS. This is because all programs in this benchmark are shorter: each program contains only 1 procedure of about 5 to 9 lines of code, while each program in the benchmark TCAS contains 8 procedures of totally 142 lines of code.

6 Related Work

There have been many approaches to repair faulty programs, and most of them use test suites to guide the repair process: they are used to localize the bug, then to generate and validate fix candidates. These approaches can be categorized into heuristic-based and semantic-based approaches. Heuristic-based tools, such as

GenProg [11,27], RSRepair [20], traverse programs' abstract syntax trees (AST) using generic programming or random search algorithms, and then modify ASTs by mutation and crossover operations. On the other hand, semantic-based tools such as SemFix [17] and Angelix [15] propose to firstly locate bug locations using ranking methods, such as Tarantula [7]. Then, they employ the symbolic execution technique to generate constraints and solve the collected constraints by a component-based repair synthesis algorithm to generate the repaired programs.

However, there is a major problem that all the test suite-based approaches need to handle is the generation of *overfitting* patches. These patches can easily pass all the test cases, but they also break untested but desired functionality of repaired programs [22]. This problem happens when the test suites, provided by users, contain concrete values, which cannot cover all the functionality of a program. To deal with this problem, the works [28–30] propose methods to automatically generate more test cases. For instance, Yang et al. [19] propose to detect overfitting patches and use fuzzing to generate more test cases to guide the repair tools. However, it is impossible to guarantee that the newly generated test cases can fully cover all behaviours of the original programs.

The works that are closer to ours are [8–10,21], which use formal specification to guide the repair process. In particular, the work [21] uses assertions to compare the output of the correct and the repaired programs. They generate the patch by performing simple code mutations, such as increasing or decreasing numerical constants by 1, or changing logical/arithmetic operators. This approach can fix simple bugs, as demonstrated by the tool AllRepair. The works [9,10] use the provided specifications to generated test cases and use a template-based approach like ours to generate the patches. Since the constraints related to the template fix are resolved by using test cases, these approaches result in many overfitting patches, as shown in our experiments with the tool Forensic.

In contrast to the test-case-based approaches, our work may does not generate overfitting patches, since the utilized specification can captures better symbolic relations of the program input/output, compared to concrete value relations in test suites. This is demonstrated in the experiments that all the patches discovered by our tool Maple are correct patches. Compared to the aforementioned specification-based approaches, our approach to generating the patches is more general. We consider the patches in the form of linear expression templates, and perform symbolic execution to collect and solve constraints over the template patches. Consequently, our tool can repair correctly more faulty programs than other specification-based tools AllRepair and Forensic.

Whereas all the above works, including ours, focus on repairing Boolean and arithmetic properties in C programs, there are works that aim to repair heap properties in C program [25,26], or repair programs in Eiffel [18], Scala [8], and C# [13]. Among these works, the tool AutoFix [18] uses test suites, the work [26] needs programmer's help, while the other works use the specification to guide the repair. Compared to ours, all these works focus on either different fragments of C programs or different programming languages. Therefore, we did not evaluate them in the experiments.

7 Limitations and Future Work

We now discuss the limitations of our work and corresponding planned improvements. There are three limitations as follows. Firstly, our current approach focuses on repairing only linear arithmetic expressions. In the future, we want to extend it to repair more types of expressions, such as arrays, strings, or dynamically allocated data structures like linked lists and trees. Secondly, our tool can fix only one expression each time. Hence, we would like to equip it with the ability of considering multiple buggy expressions at the same time. Thirdly, the tool cannot synthesize missing expressions. Since specifications are used, this problem is equivalent to finding correct code fragments that meet the specifications of the missing expressions. Thus, we can follow the approach of [25] which learns specifications of the existing programs to find the removed program fragments.

8 Conclusions

We have introduced an automated program repair framework using formal verification and expression templates. More specifically, we first utilize a formal verification system to locate and rank the potentially buggy expressions by their likelihood to cause the bug. Then, each buggy expression is replaced by a template patch, which is a linear expression of the program's variables with unknown coefficients, to create a template program. This program will be verified against to collect constraints of the template patch. Finally, we apply a constraint solving technique using Farkas' lemma to solve these constraints to discover the repaired program. In practice, our prototype tool Maple can discover more correct patches than other program repair tools in the widely used benchmark TCAS. It can also fix many challenging programs in the synthetic benchmark Recursion, which cannot be fully repaired by other tools.

Acknowledgements. We are grateful to the anonymous reviewers for their valuable feedback. The first author would like to thank Bat-Chen Rothenberg for her help on the experimentation of AllRepair, and Xianglong Kong for sharing unit tests of the TCAS benchmark. This work is supported by the NRF grant NRF2014NCR-NCR001-030.

References

1. Chin, W.-N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. (SCP) **77**(9), 1006–1036 (2012)
2. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_39
3. Do, H., Elbaum, S.G., Rothermel, G.: Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empirical Softw. Eng. **10**(4), 405–435 (2005)

4. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
5. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning About Systems. Cambridge University Press, New York (2004). ISBN 052154310X
6. ISO. ISO/IEC 9899: 2011 information technology - programming languages - C. Int. Organ. Stand. **27**, 59 (2011)
7. Jones, J.A., Harrold, M.J., Stasko, J.T.: Visualization of test information to assist fault localization. In: International Conference on Software Engineering (ICSE), pp. 467–477 (2002)
8. Kneuss, E., Koukoutos, M., Kuncak, V.: Deductive program repair. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 217–233. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_13
9. Könighofer, R., Bloem, R.: Automated error localization and correction for imperative programs. In: International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 91–100 (2011)
10. Könighofer, R., Bloem, R.: Repair with on-the-fly program analysis. In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS, vol. 7857, pp. 56–71. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39611-3_11
11. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: a generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2012)
12. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
13. Logozzo, F., Ball, T.: Modular and verified automatic program repair. In: International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 133–146 (2012)
14. Long, F., Rinard, M.: Staged program repair with condition synthesis. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 166–178 (2015)
15. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: scalable multiline program patch synthesis via symbolic analysis. In: International Conference on Software Engineering (ICSE), pp. 691–701 (2016)
16. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
17. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: SemFix: program repair via semantic analysis. In: International Conference on Software Engineering (ICSE), pp. 772–781 (2013)
18. Pei, Y., Furia, C.A., Nordio, M., Wei, Y., Meyer, B., Zeller, A.: Automated fixing of programs with contracts. IEEE Trans. Softw. Eng. **40**(5), 427–449 (2014)
19. Program Repair Website (2018). http://program-repair.org/. Accessed 18 Sept 2018
20. Qi, Y., Mao, X., Lei, Y., Dai, Z., Wang, C.: The strength of random search on automated program repair. In: International Conference on Software Engineering (ICSE), pp. 254–265 (2014)
21. Rothenberg, B.-C., Grumberg, O.: Sound and complete mutation-based program repair. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 593–611. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_36

22. Smith, E.K., Barr, E.T., Le Goues, C., Brun, Y.: Is the cure worse than the disease? Overfitting in automated program repair. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 532–543 (2015)
23. Ta, Q.-T., Le, T.C., Khoo, S.C., Chin, W.N.: Automated lemma synthesis in symbolic-heap separation logic. In: Symposium on Principles of Programming Languages (POPL), pp. 9:1–9:29 (2018)
24. Ta, Q.-T., Le, T.C., Khoo, S.-C., Chin, W.-N.: Automated mutual explicit induction proof in separation logic. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 659–676. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_40
25. van Tonder, R., Le Goues, C.: Static automated program repair for heap properties. In: International Conference on Software Engineering (ICSE), pp. 151–162 (2018)
26. Verma, S., Roy, S.: Synergistic debug-repair of heap manipulations. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 163–173 (2017)
27. Weimer, W., Forrest, S., Le Goues, C., Nguyen, T.: Automatic program repair with evolutionary computation. Commun. ACM **53**(5), 109–116 (2010)
28. Xin, Q., Reiss, S.P.: Identifying test-suite-overfitted patches through test case generation. In: International Symposium on Software Testing and Analysis, pp. 226–236 (2017)
29. Yang, J., Zhikhartsev, A., Liu, Y., Tan, L.: Better test cases for better automated program repair. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 831–841 (2017)
30. Yu, Z., Martinez, M., Danglot, B., Durieux, T., Monperrus, M.: Test case generation for program repair: a study of feasibility and effectiveness. CoRR abs/1703.00198 (2017)