Connecting Program Synthesis and Reachability:
Automatic Program Repair using Test-Input Generation

ThanhVu Nguyen[1], Westley Weimer[2], Deepak Kapur[3], and Stephanie Forrest[3]

[1] University of Nebraska, Lincoln NE, USA, tnguyen@cse.unl.edu
[2] University of Virginia, Charlottesville VA, USA, weimer@virginia.edu
[3] University of New Mexico, Albuquerque NM, USA, {kapur,forrest}@cs.unm.edu

Abstract. We prove that certain formulations of program synthesis and reachability are equivalent. Specifically, our constructive proof shows the reductions between the template-based synthesis problem, which generates a program in a pre-specified form, and the reachability problem, which decides the reachability of a program location. This establishes a link between the two research fields and allows for the transfer of techniques and results between them.

To demonstrate the equivalence, we develop a program repair prototype using reachability tools. We transform a buggy program and its required specification into a specific program containing a location reachable only when the original program can be repaired, and then apply an off-the-shelf test-input generation tool on the transformed program to find test values to reach the desired location. Those test values correspond to repairs for the original program. Preliminary results suggest that our approach compares favorably to other repair methods.

Keywords: program synthesis; program verification; program reachability; reduction proof; automated program repair; test-input generation;

1 Introduction

Synthesis is the task of generating a program that meets a required specification. Verification is the task of validating program correctness with respect to a given specification. Both are long-standing problems in computer science, although there has been extensive work on program verification and comparatively less on program synthesis until recently. Over the past several years, certain verification techniques have been adopted to create programs, e.g., applying symbolic execution to synthesize program repairs [27, 28, 31, 33], suggesting the possibility that these two problems may be “two sides of the same coin”. Finding and formalizing this equivalence is valuable in both theory and practice: it allows comparisons between the complexities and underlying structures of the two problems, and it raises the possibility of additional cross-fertilization between two fields that are usually treated separately (e.g., it might enable approximations designed to solve one problem to be applied directly to the other).

This paper establishes a formal connection between certain formulations of program synthesis and verification. We focus on the *template-based synthesis* problem, which generates missing code for partially completed programs, and we view verification as a *reachability* problem, which checks if a program can reach an undesirable state. We then prove constructively that template-based synthesis and reachability are *equivalent*. We reduce a template-based synthesis problem, which consists of a program with parameterized templates to be synthesized and a test suite specification, to a program consisting of a specific location that is reachable only when those templates can be instantiated such that the program meets the given specification. To reduce reachability to synthesis, we transform a reachability instance consisting of a program and a given location into a synthesis instance that can be solved only when the location in the original problem is reachable. Thus, reachability solvers can be applied to synthesize code, and conversely, synthesis tools can be used to determine reachability.

To demonstrate the equivalence, we use the reduction to develop a new automatic program repair technique using an existing test-input generation tool. We view *program repair* as a special case of template-based synthesis in which "patch" code is generated so that it behaves correctly. We present a prototype tool called CETI that automatically repairs C programs that violate test-suite specifications. Given a test suite and a program failing at least one test in that suite, CETI first applies fault localization to obtain a list of ranked suspicious statements from the buggy program. For each suspicious statement, CETI transforms the buggy program and the information from its test suite into a program reachability instance. The reachability instance is a new program containing a special `if` branch, whose `then` branch is reachable only when the original program can be repaired by modifying the considered statement. By construction, any input value that allows the special location to be reached can map directly to a repair template instantiation that fixes the bug. To find a repair, CETI invokes an off-the-shelf automatic test-input generation tool on the transformed code to find test values that can reach the special branch location. These values correspond to changes that, when applied to the original program, cause it to pass the given test suite. This procedure is guaranteed to be sound, but it is not necessarily complete. That is, there may be bugs that the procedure cannot find repairs for, but all proposed repairs are guaranteed to be correct with respect to the given test suite. We evaluated CETI on the `Tcas` program [14], which has 41 seeded defects, and found that it repaired over 60%, which compares favorably with other state-of-the-art automated bug repair approaches.

To summarize, the main contributions of the paper include:

- *Equivalence Theorem*: We constructively prove that the problems of template-based program synthesis and reachability in program verification are equivalent. Even though these two problems are shown to be undecidable in general, the constructions allow heuristics solving one problem to be applied to the other.
- *Automatic Program Repair*: We present a new automatic program repair technique, which leverages the construction. The technique reduces the task of synthesizing program repairs to a reachability problem, where the results

```
1  int is_upward(int in,int up,int down){
2    int bias, r;
3    if (in)
4      bias = down; //fix: bias = up + 100
5    else
6      bias = up;
7    if (bias > down)
8      r = 1;
9    else
10     r = 0;
11   return r;
12 }
```

Test	Inputs			Output		Passed?
	in	up	down	expected	observed	
1	1	0	100	0	0	✓
2	1	11	110	1	0	✗
3	0	100	50	1	1	✓
4	1	-20	60	1	0	✗
5	0	0	10	0	0	✓
6	0	0	-10	1	1	✓

Fig. 1. Example buggy program and test suite. CETI suggests replacing line 4 with the statement `bias = up + 100;` to repair the bug.

produced by a test-input generation tool correspond to a patch that repairs the original program.

– *Implementation and Evaluation*: We implement the repair algorithm in a prototype tool that automatically repairs C programs, and we evaluate it on a benchmark that has been targeted by multiple program repair algorithms.

2 Motivating Example

We give a concrete example of how the reduction from template-based synthesis to reachability can be used to repair a buggy program. Consider the buggy code shown in Figure 1, a function excerpted from a traffic collision avoidance system [14]. The intended behavior of the function can be precisely described as: `is_upward(in,up,down) = in*100 + up > down`. The table in Figure 1 gives a test suite describing the intended behavior. The buggy program fails two of the tests, which we propose to repair by synthesizing a patch.

We solve this synthesis problem by restricting ourselves to generating patches under predefined templates, e.g., synthesizing expressions involving program variables and unknown parameters, and then transforming this template-based synthesis problem into a reachability problem instance. In this approach, a template such as

$$\boxed{c_0} + \boxed{c_1} v_1 + \boxed{c_2} v_2$$

is a linear combination[4] of program variables v_i and unknown template parameters $\boxed{c_i}$. For clarity, we often denote template parameters with a box to distinguish them from normal program elements. This template can be instantiated to yield concrete expressions such as $200 + 3v_1 + 4v_2$ via $c_0 = 200, c_1 = 3, c_2 = 4$. To repair Line 4 of Figure 1, (`bias = down;`) with a linear template, we would replace Line 4 with:

`bias =` $\boxed{c_0}$ `+`$\boxed{c_1}$`*bias +`$\boxed{c_2}$`*in +`$\boxed{c_3}$`*up +`$\boxed{c_4}$`*down;`

[4] More general templates (e.g., nonlinear polynomials) are also possible as shown in Section 3.4.

```
int c0,c1,c2,c3,c4; //global inputs

int is_upward_P(int in,int up,int
    down){
  int bias, r;
  if (in)
    bias =
     c0+c1*bias+c2*in+c3*up+c4*down;
  else
    bias = up;
  if (bias > down)
     r = 1;
  else
     r = 0;
  return r;
}

int main() {
   if(is_upward_P(1,0,100) == 0 &&
      is_upward_P(1,11,110) == 1 &&
      is_upward_P(0,100,50) == 1 &&
      is_upward_P(1,-20,60) == 1 &&
      is_upward_P(0,0,10) == 0 &&
      is_upward_P(0,0,-10) == 1){
     [L]
   }
   return 0;
}
```

Fig. 2. The reachability problem instance derived from the buggy program and test suite in Figure 1. Location L is reachable with values such as $c_0 = 100, c_1 = 0, c_2 = 0, c_3 = 1, c_4 = 0$. These values suggest using the statement `bias = 100 + up;` at Line 4 in the buggy program.

where `bias`, `in`, `up`, and `down` are the variables in scope at Line 4 and the value of each c_i must be found. We propose to find them by constructing a special program reachability instance and then solving that instance.

The construction transforms the program, its test suite (Figure 1), and the template statement into a reachability instance consisting of a program and target location. The first key idea is to derive a new program containing the template code with the template parameters $\boxed{c_i}$ represented explicitly as program variables c_i. This program defines the reachability instance, which must assign values to each c_i. The second key idea is that each test case is explicitly represented as a conditional expression. Recall that we seek a single synthesis solution (one set of values for c_i) that respects all tests. Each test is encoded as a conditional expression (a reachability constraint), and we take their conjunction, being careful to refer to the same c_i variables in each expression. In the example, we must find one repair that satisfies all six tests, not six separate repairs that each satisfy only one test.

The new program, shown in Figure 2, contains a function `is_upward`$_P$ that resembles the function `is_upward` in the original code but with Line 4 replaced by the template statement with each reference to a template parameter replaced by a reference to the corresponding new externally-defined program variable. The program also contains a starting function `main`, which encodes the inputs and expected outputs from the given test suite as the guards to the conditional statement leading to the target location L. Intuitively, the reachability problem instance asks if we can find values for each c_i that allow control flow to reach location L, which is only reachable iff all tests are satisfied.

This reachability instance can be given as input to any off-the-self test-input generation tool. Here, we use KLEE [9] to find value for each c_i. KLEE determines that the values $c_0 = 100, c_1 = 0, c_2 = 0, c_3 = 1, c_4 = 0$ allow control flow to reach location L. Finally, we map this solution back to the original program repair problem by applying the c_i values to the template bias = $\boxed{c_0}$ +$\boxed{c_1}$*bias +$\boxed{c_2}$*in +$\boxed{c_3}$*up +$\boxed{c_4}$*down;

generating the statement:

```
bias = 100 + 0*bias + 0*in + 1*up + 0*down;
```

which reduces to `bias = 100 + up`. Replacing the statement `bias = down` in the original program with the new statement `bias = 100 + up` produces a program that passes all of the test cases.

To summarize, a specific question (i.e., can the bug be repaired by applying template X to line Y of program P while satisfying test suite T?) is reduced to a single reachability instance, solvable using a reachability tool such as a test-input generator. This reduction is formally established in the next section.

3 Connecting Program Synthesis and Reachability

We establish the connection between the template-based formulation of program synthesis and the reachability problem in program verification. We first review these problems and then show their equivalence.

3.1 Preliminaries

We consider standard imperative programs in a language like C. The base language includes usual program constructs such as assignments, conditionals, loops, and functions. A function takes as input a (potentially empty) tuple of values and returns an output value. A function can call other functions, including itself. For simplicity, we equate a program P with its finite set of functions, including a special starting function main_P. For brevity, we write $P(c_i, \ldots, c_n) = y$ to denote that evaluating the function $\text{main}_P \in P$ on the input tuple $(c_i, \ldots, c_n)$ results in the value y. Program or function semantics are specified by a test suite consisting of a finite set of input/output pairs. When possible, we use c_i for concrete input values and v_i for formal parameters or variable names.

To simplify the presentation, we assume that the language also supports exceptions, admitting nonlocal control flow by raising and catching exceptions as in modern programming languages such as C++ and Java. We discuss how to remove this assumption in Section 3.3.

Template-based Program Synthesis. *Program synthesis* aims to automatically generate program code to meet a required specification. The problem of synthesizing a complete program is generally undecidable [44], so many practical synthesis techniques operate on partially-complete programs, filling in well-structured gaps [43, 45, 41, 38, 2, 22, 1, 46]. These techniques synthesize programs from specific grammars, forms, or templates and do not generate arbitrary code. A synthesis *template* expresses the shape of program constructs, but includes holes (sometimes called template parameters), as illustrated in the previous section. We refer to a program containing such templates as a *template program* and extend the base language to include a finite, fixed set of template parameters $\boxed{c_i}$ as shown earlier. Using the notation of contextual operational semantics, we write $P[c_0, \ldots, c_n]$ to denote the result of instantiating the template program P with template parameter values $c_0 \ldots c_n$. To find values for the parameters in a template program, many techniques (e.g., [43, 45, 1, 46]) encode the program and its specification as a logical formula (e.g., using axiomatic semantics) and use a constraint solver such as SAT or SMT to find values for the parameters c_i that satisfy the formula. Instantiating the templates with those values produces a complete program that adheres to the required specification.

Definition 1. Template-based Program Synthesis Problem. *Given a template program Q with a finite set of template parameters $S = \{\boxed{c_1}, \ldots, \boxed{c_n}\}$ and a finite test suite of input/output pairs $T = \{(i_1, o_1), \ldots, (i_m, o_m)\}$, do there exist parameter values c_i such that $\forall (i, o) \in T \; . \; (Q[c_1, \ldots, c_n])(i) = o$?*

For example, the program in Figure 1 with Line 4 replaced by `bias =` $\boxed{c_0}$ `+`$\boxed{c_1}$`*bias +`$\boxed{c_2}$`*in +`$\boxed{c_3}$`*up +`$\boxed{c_4}$`*down` is an instance of template-based synthesis. This program passes its test suite given in Figure 1 using the solution $\{c_0 = 100, c_1 = 1, c_2 = 0, c_3 = 1, c_4 = 0\}$. The decision formulation of the problem asks if satisfying values $c_1 \ldots c_n$ exist; in this presentation we require that witnesses be produced.

Program Reachability. *Program reachability* is a classic problem which asks if a particular program state or location can be observed at run-time. It is not decidable in general, because it can encode the halting problem (cf. Rice's Theorem [37]). However, reachability remains a popular and well-studied verification problem in practice. In model checking [11], for example, reachability is used to determine whether program states representing undesirable behaviors could occur in practice. Another application area is test-input generation [10], which aims to produce test values to explore all reachable program locations.

Definition 2. Program Reachability Problem. *Given a program P, set of program variables $\{x_1, \ldots, x_n\}$ and target location L, do there exist input values c_i such that the execution of P with x_i initialized to c_i reaches L in a finite number of steps?*

For example, the program in Figure 3 has a reachable location L using the solution $\{x = -20, y = -40\}$. Similar to the synthesis problem, the decision problem formulation of reachability merely asks if the input values $c_1, \ldots, c_n$ exist; in this presentation we require witnesses be produced.

3.2 Reducing Synthesis to Reachability

We present the constructive reduction from synthesis to reachability. The key to the reduction is a particular "gadget", which constructs a reachability instance that can be satisfied iff the synthesis problem can be solved.

```
//global inputs
int x, y;

int P(){
  if (2 * x == y)
    if (x > y + 10)
      [L]

  return 0;
}
```

Fig. 3. An instance of program reachability. Program P reaches location L using the solution $\{x = -20, y = -40\}$.

```
int P_Q() {
  if (2*[x] == [y])
   if([x] > [y]+10)
     //loc L in P
     raise
         REACHED;

  return 0;
}
```

```
int main_Q() {
  //synthesize x, y
  int x = c_x;
  int y = c_y;
  try
    P_Q();
  catch (REACHED)
    return 1;

  return 0;
}
```

Fig. 4. Reducing the reachability example in Figure 3 to a template-based program synthesis program (i.e., synthesize assignments to c_x and c_y). The test suite of the reduced synthesis program is $Q() = 1$.

Reduction: Let Q be a template program with a set of template parameters $S = \{\boxed{c_1}, \ldots, \boxed{c_n}\}$ and a set of finite tests $T = \{(i_1, o_1), \ldots\}$. We construct $\mathsf{GadgetS2R}(Q, S, T)$, which returns a new program P (the constructed reachability instance) with a special location L, as follows:

1. For every template parameter $\boxed{c_i}$, add a fresh global variable v_i. A solution to this reachability instance is an assignment of concrete values c_i to the variables v_i.
2. For every function $q \in Q$, define a similar function $q_P \in P$. The body of q_P is the same as q, but with every reference to a template parameter $\boxed{c_i}$ replaced with a reference to the corresponding new variable v_i.
3. P also contains a starting function main_P that encodes the specification information from the test suite T as a conjunctive expression e:

$$e = \bigwedge_{(i,o) \in T} \text{main}_{QP}(i) = o$$

   where main_{QP} is a function in P corresponding to the starting function main_Q in Q. In addition, the body of main_P is one conditional statement leading to a fresh target location L if and only if e is true. Thus, main_P has the form

```
int main_P() {
  if (e)
    [L]
}
```

4. The derived program P consists of the declaration of the new variables (Step 1), the functions q_P's (Step 2), and the starting function main_P (Step 3).

Example: Figure 2 illustrates the reduction using the example from Figure 1. The resulting reachability program can arrive at location L using the input $\{c_0 = 100, c_1 = 0, c_2 = 0, c_3 = 1, c_4 = 0\}$, which corresponds to a solution.

Reduction Correctness and Complexity: The correctness of GadgetS2R, which transforms synthesis to reachability, relies on two key invariants. First, function calls in the derived program P have the same behavior as template functions in the original program Q. Second, location L is reachable if and only if values c_i can be assigned to variables v_i such that Q passes all of the tests. The full correctness proof is given in [35].

The complexity of GadgetS2R is *linear* in both the program size and number of test cases of the input instance Q, S, T. The constructed program P consists of all functions in Q (with $|S|$ extra variables) and a starting function main_P with an expression encoding the test suite T.

This reduction directly leads to the main result for this direction of the equivalence:

Theorem 1. *The template-based synthesis problem in Definition 1 is reducible to the reachability problem in Definition 2.*

3.3 Reducing Reachability to Synthesis

Here, we present the reduction from reachability to synthesis. The reduction also uses a particular gadget to construct a synthesis instance that can be solved iff the reachability instance can be determined.

Reduction: Let P be a program, L be a location in P, and $V = \{v_1, \ldots, v_n\}$ be global variables never directly assigned in P. We construct GadgetR2S(P, L, V), which returns a template program Q with template parameters S and a test suite T, as follows:

1. For every variable v_i, define a fresh template variable $\boxed{c_i}$. Let the set of template parameters S be the set containing each $\boxed{c_i}$.
2. For every function $p \in P$, define a derived function $p_Q \in Q$. Replace each function call to p with the corresponding call to p_Q. Replace each use of a variable v_i with a read from the corresponding template parameter $\boxed{c_i}$; remove all declarations of variables v_i.
3. Raise a unique exception `REACHED`, at the location in Q corresponding to the location L in P. As usual, when an exception is raised, control immediately jumps to the most recently-executed *try-catch* block matching that exception. The exception `REACHED` will be caught iff the location in Q corresponding to $L \in P$ would be reached.
4. Define a starting function main_Q that has no inputs and returns an integer value. Let main_{PQ} be the function in Q corresponding to the starting function main_P in P.
   - Insert *try-catch* construct that calls p_Q and returns the value 1 if the exception `REACHED` is caught.
   - At the end of main_Q, return the value 0.
   - Thus, main_Q has the form

```
int main_Q() {
  try {
    main_PQ();
  } catch (REACHED) {
    return 1;
  }
  return 0;
}
```

5. The derived program Q consists of the finite set of template parameters $S = \{\boxed{c_1}), \ldots, \boxed{c_n}\}$ (Step 1), functions p_Q's (Step 2), and the starting function main_Q (Step 4).
6. The test suite T for Q consists of exactly one test case $Q() = 1$, indicating the case when the exception `REACHED` is raised and caught.

Example: Figure 4 illustrates the reduction using the example from Figure 3. The synthesized program can be satisfied by $c_0 = -20, c_1 = -40$, corresponding to the input $(x = -20, y = -40)$ which reaches L in Figure 3.

The exception `REACHED` represents a unique signal to main_Q that the location L has been reached. Many modern languages support exceptions for handling special events, but they are not strictly necessary for the reduction to succeed. Other (potentially language-dependent) implementation techniques could also be employed. Or, we could use a tuple to represent the signal, e.g., returning (v, false) from a function that normally returns v if the location corresponding L has not been reached and $(1, \mathsf{true})$ as soon as it has. BLAST [7], a model checker for C programs (which do not support exceptions), uses *goto* and labels to indicate when a desired location has been reached.

Reduction Correctness and Complexity: The correctness of the GadgetS2R, which transforms reachability to synthesis, depends on two key invariants. First, for any c_i, execution in the derived template program Q with $\boxed{c_i} \mapsto c_i$ mirrors execution in P with $v_i \mapsto c_i$ up to the point when L is reached (if ever). Second, the exception `REACHED` is raised in Q iff location L is reachable in P. The full correctness proof is given in [35]

The complexity of GadgetR2S is *linear* in the input instance P, L, v_i. The constructed program Q consists of all functions in P and a starting function $\text{main}Q$ having n template variables, where $n = |\{v_i\}|$.

This reduction directly leads to the main result for this direction of the equivalence:

Theorem 2. *The reachability problem in Definition 2 is reducible to the template-based synthesis problem in Definition 1.*

3.4 Synthesis ≡ Reachability

Together, the above two theorems establish the equivalence between the reachability problem in program verification and the template-based program synthesis.

Corollary 1. *The reachability problem in Definition 2 and the template-based synthesis problem in Definition 1 are linear-time reducible to each other.*

This equivalence is perhaps unsurprising as researchers have long assumed certain relations between program synthesis and verification (e.g., see Related Work in Section 5). However, we believe that a proof of the equivalence is valuable. First, our constructive proof, although straightforward, formally shows that both problems inhabit the same complexity class (e.g., the restricted formulation of synthesis in Definition 1 is as hard as the reachability problem in Definition 2). Second, although both problems are undecidable in the general case, the linear-time transformations allow existing approximations and ideas developed for one problem to apply to the other one. Third, in term of practicality, the equivalence allows for direct application of off-the-shelf reachability and verification tools to synthesize and repair programs. Our approach is not so different from verification works that transform the interested problems into e.g., SAT/SMT formulas to be solved by existing efficient solvers. Finally, this work can be extended to more complex classes of synthesis and repair problems. For example, we demonstrate the approach using linear templates but more general templates such as combinations of nonlinear polynomials can be handled via a priority subset of terms (e.g., $t_1 = x^2, t_2 = xy$, as demonstrated in nonlinear invariant generation [34]). Also, instead of using finite tests, we could extend the definition and reduction to synthesize and repair programs with respect to formal specifications such as assertions and pre/post conditions.

We hope that these results help raise fruitful cross-fertilization among program verification and synthesis fields that are usually treated separately. Because our reductions result in particular reachability problem instances (e.g., with large guards) that are rarely encountered by current verification techniques, they may help refine existing tools or motivate optimization in new directions. As an example, our bug repair prototype CETI (discussed in the next Section) has produced reachability instances that hit a bug in KLEE, which crashes the tool and is confirmed to be important by the developers[5]. These hard instances might be used to evaluate and improve verification and synthesis tools (similar to benchmarks used in annual SAT[6] and SMT[7] competitions).

4 Automatic Program Repair using Test-Input Generation

We use the equivalence to develop CETI (Correcting Errors using Test Inputs), a tool for automated program repair (a synthesis problem) using test-input generation techniques (which solves reachability problems). We define problem of program repair in terms of template-based program synthesis:

Definition 3. Program Repair Problem. *Given a program Q that fails at least one test in a finite test suite T and a finite set of parameterized templates S, does there exist a set of statements $\{s_i\} \subseteq Q$ and parameter values $c_1, \ldots, c_n$*

[5] http://mailman.ic.ac.uk/pipermail/klee-dev/2016-February/001278.html

[6] SAT Competitions: http://www.satcompetition.org

[7] SMT competitions: http://smtcomp.sourceforge.net/2016

for the templates in S such that s_i can be replaced with $S[c_1, \ldots, c_n]$ and the resulting program passes all tests in T?

This repair problem thus allows edits to multiple program statements (e.g., we can replace both lines 4 and 10 in Figure 1 with parameterized templates). The *single-edit* repair problem restricts the edits to one statement.

CETI implements the key ideas from Theorem 1 in Section 3.2 to transform this repair problem into a reachability task solvable by existing verification tools. Given a test suite and a buggy program that fails some test in the suite, CETI employs the statistical fault localization technique Tarantula [25] to identify particular code regions for synthesis, i.e., program statements likely related to the defect. Next, for each suspicious statement and synthesis template, CETI transforms the buggy program, the test suite, the statement and the template into a new program containing a location reachable only when the original program can be repaired. Thus, by default CETI considers single-edit repairs, but it can be modified to repair multiple lines by using k top-ranked suspicious statements.

Our current implementation uses repair templates similar to those given in [27, 33], which allow modifying constants, expressions (such as the linear template shown in Section 2), and logical, comparisons, and arithmetic operators (such as changing $\|\|$ to &&, $\leq$ to $<$, or $+$ to $-$). Finally, we send the transformed program to the test-input generation tool KLEE, which produces test values that can reach the designated location. Such test input values, when combined with the synthesis template and the suspicious statement, correspond exactly to a patch that repairs the bug. CETI synthesizes correct-by-construction repairs, i.e., the repair, if found, is guaranteed to pass the test suite.

4.1 Evaluation

To evaluate CETI, we use the `Tcas` program from the SIR benchmark [14]. The program, which implements an aircraft traffic collision avoidance system, has 180 lines of code and 12 integer inputs. The program comes with a test suite of about 1608 tests and 41 faulty functions, consisting of seeded defects such as changed operators, incorrect constant values, missing code, and incorrect control flow. Among the programs in SIR, `Tcas` has the most introduced defects (41), and it has been used to benchmark modern bug repair techniques [13, 28, 33].

We manually modify `Tcas`, which normally prints its result on the screen, to instead return its output to its caller, e.g., `printf("output is %d\n",v)` becomes `return v`. For efficiency, many repair techniques initially consider a smaller number of tests in the suite and then verify candidate repairs on the entire suite [33]. In contrast, we use all available tests at all times to guarantee that any repair found by CETI is correct with respect to the test suite. We find that modern tools such as KLEE can handle the complex conditionals that encode such information efficiently and generate the desired solutions within seconds.

The behavior of CETI is controlled by customizable parameters. For the experiments described here, we consider the top $n = 80$ statements with a score

$s > 0.2$ from the ranked list of suspicious statements and, then, apply the predefined templates to these statements. For efficiency, we restrict synthesis parameters to be within certain value ranges: constant coefficients c_0 are confined to the integral range $[-100000, 100000]$ while the variable coefficients c_1, c_2 are drawn from the set $\{-1, 0, 1\}$.

	Bug Type	**R-Progs**	**T(s)**	**Repair?**		**Bug Type**	**R-Progs**	**T(s)**	**Repair?**
v1	incorrect op	6143	21	✓	v22	missing code	5553	175	–
v2	missing code	6993	27	✓	v23	missing code	5824	164	–
v3	incorrect op	8006	18	✓	v24	missing code	6050	231	–
v4	incorrect op	5900	27	✓	v25	incorrect op	5983	19	✓
v5	missing code	8440	394	–	v26	missing code	8004	195	–
v6	incorrect op	5872	19	✓	v27	missing code	8440	270	–
v7	incorrect const	7302	18	✓	v28	incorrect op	9072	11	✓
v8	incorrect const	6013	19	✓	v29	missing code	6914	195	–
v9	incorrect op	5938	24	✓	v30	missing code	6533	170	–
v10	incorrect op	7154	18	✓	v31	multiple	4302	16	✓
v11	multiple	6308	123	–	v32	multiple	4493	17	✓
v12	incorrect op	8442	25	✓	v33	multiple	9070	224	–
v13	incorrect const	7845	21	✓	v34	incorrect op	8442	75	✓
v14	incorrect const	1252	22	✓	v35	multiple	9070	184	–
v15	multiple	7760	258	–	v36	incorrect const	6334	10	✓
v16	incorrect const	5470	19	✓	v37	missing code	7523	174	–
v17	incorrect const	7302	12	✓	v38	missing code	7685	209	–
v18	incorrect const	7383	18	✓	v39	incorrect op	5983	20	✓
v19	incorrect const	6920	19	✓	v40	missing code	7364	136	–
v20	incorrect op	5938	19	✓	v41	missing code	5899	29	✓
v21	missing code	5939	31	✓					

Table 1. Repair Results for 41 **Tcas** Defects

Results. Table 1 shows the experimental results with 41 buggy `Tcas` versions. These experiments were performed on a 32-core 2.60GHz Intel Linux system with 128 GB of RAM. Column **Bug Type** describes the type of defect. *Incorrect Const* denotes a defect involving the use of the wrong constant, e.g., 700 instead of 600. *Incorrect Op* denotes a defect that uses the wrong operator for arithmetic, comparison, or logical calculations, e.g., $\geq$ instead of $>$. *Missing code* denotes defects that entirely lack an expression or statement, e.g., $a\&\&b$ instead of $a\&\&b||c$ or `return a` instead of `return a+b`. *Multiple* denotes defects caused by several actions such as missing code at a location and using an incorrect operator at another location. Column **T(s)** shows the time taken in seconds. Column **R-Prog** lists the number of reachability program instances that were generated and processed by KLEE. Column **Repair?** indicates whether a repair was found.

We were able to correct 26 of 41 defects, including multiple defects of different types. On average, CETI takes 22 seconds for each successful repair. The tool found 100% of repairs for which the required changes are single edits according to one of our predefined templates (e.g., generating arbitrary integer constants or changing operators at one location). In several cases, defects could be repaired in several ways. For example, defect v_{28} can be repaired by swapping the results of both branches of a conditional statement or by inverting the conditional guard. CETI also obtained unexpected repairs. For example, the bug in v_{13} is a comparison against an incorrect constant; the buggy code reads `< 700` while the human-written patch reads `< 600`. Our generated repair of `< 596` also passes all tests.

We were not able to repair 15 of 41 defects, each of which requires edits at multiple locations or the addition of code that is beyond the scope of the current set of templates. As expected, CETI takes longer for these programs because it tries all generated template programs before giving up. One common pattern among these programs is that the bug occurs in a macro definition, e.g., `#define C = 100` instead of `#define C = 200`. Since the CIL front end automatically expands such macros, CETI would need to individually fix each use of the macro in order to succeed. This is an artifact of CIL, rather than a weakness inherent in our algorithm.

CETI, which repairs 26 of 41 `Tcas` defects, performs well compared to other reported results from repair tools on this benchmark program. GenProg, which finds edits by recombining existing code, can repair 11 of these defects [33, Tab. 5]. The technique of Debroy and Wong, which uses random mutation, can repair 9 defects [13, Tab. 2]. FoREnSiC, which uses the concolic execution in CREST, repairs 23 defects [28, Tab. 1]. SemFix out-performs CETI, repairing 34 defects [33, Tab. 5], but also uses fifty test cases instead of the entire suite of thousands[8]. Other repair techniques, including equivalence checking [28] and counterexample guided refinement [28], repair 15 and 16 defects, respectively.

Although CETI uses similar repair templates as both SemFix and FoREnSiC, the repair processes are different. SemFix directly uses and customizes the KLEE symbolic execution engine, and FoRenSiC integrates concolic execution to analyze programs and SMT solving to generate repairs. In contrast, CETI eschews heavyweight analyses, and it simply generates a reachability instance. Indeed, our work is inspired by, and generalizes, these work, observing that the whole synthesis task can be offloaded with strong success in practice.

However, there is a trade-off: customizing a reachability solver to the task of program repair may increase the performance or the number of repairs found, but may also reduce the generality or ease-of-adoption of the overall technique. We note that our unoptimized tool CETI already outperforms published results for GenProg, Debroy and Wong, and FoREnSiC on this benchmark, and is competitive with SemFix.

[8] Thus CETI's repairs, which pass the entire suite instead of just 50 selected tests, meet a higher standard. We were unable to obtain SemFix details, e.g., which 50 tests, online or from the authors.

Limitations. We require that the program behaves deterministically on the test cases and that the defect be reproducible. This limitation can be mitigated by running the test cases multiple times, but ultimately our technique is not applicable if the program is non-deterministic. We assume that the test cases encode all relevant program requirements. If adequate test cases are not available then the repair may not retain required functionality. Our formulation also encodes the test cases as inputs to a starting function (e.g., `main`) with a single expected output. This might not be feasible for certain types of specifications, such as liveness properties ("eventually" and "always") in temporal logic. The efficiency of CETI depends on fault localization to reduce the search space. The reachability or test-input generation tool used affects both the efficiency and the efficacy of CETI. For example, if the reachability tool uses a constraint solver that does not support data types such as string or arrays then we will not be able to repair program defects involving those types. Finally, we assume that the repair can be constructed from the provided repair templates.

The reduction in Section 3.2 can transform a finite space (buggy) program into an infinite space reachability problem (e.g., we hypothesize that a bounded loop guard $i \leq 10$ is buggy and try to synthesize a new guard using an unknown parameter $i \leq \boxed{c}$). However, this does not invalidate the theoretical or empirical results and the transformation is efficient in the program size and the number of tests. The transformation also might not be optimal if we use complex repair templates (e.g., involving many unknown parameters). In practice we do not need to synthesize many complex values and thus modern symbolic execution tools such as KLEE can solve the transformed problems efficiently, as shown in our evaluation.

This paper concretely demonstrates the applicability of program reachability (test-input generation) to program synthesis (defect repair) but not the reverse direction of using program synthesis to solve reachability. Applying advances in automatic program repair to find test-inputs to reach nontrivial program locations remains future work.

5 Related Work

Program Synthesis and Verification. Researchers have long hypothesized about the relation between program synthesis and verification and proposed synthesis approaches using techniques or tools often used to verify programs such as constraint solving or model checking [1, 45]. For example, Bodik and Solar-Lezama et. al.'s work [42, 41] on sketching defines the synthesis task as: $\exists c \,.\, \forall (i, o) \,.\, \in T \,.\, (P[c])(i) = o$ (similar to our template-based synthesis formulation in Definition 1) and solves the problem using a SAT solver. Other synthesis and program repair researches, e.g., [46, 45, 33, 31, 5], also use similar formulation to integrate verification tools, e.g., test-input generation, to synthesize desired programs. In general, such integrations are common in many ongoing synthesis works including the multi-disciplinary ExCAPE project [15] and the SyGuS competition [47], and have produced many practical and useful tools such as

Sketch that generates low-level bit-stream programs [41], Autograder that provides feedbacks on programming homework [40], and FlashFill that constructs Excel macros [20, 21].

The work presented in this paper is inspired by these works, and generalizes them by establishing a formal connection between synthesis and verification using the template-based synthesis and reachability formulations. We show that it is not just a coincident that the aforementioned synthesis works can exploit verification techniques, but that every template-based synthesis problem can be reduced to the reachability formulation in verification. Dually, we show the other direction that reduces reachability to template-based synthesis, so that every reachability problem can be solved using synthesis. Furthermore, our constructive proofs describe efficient algorithms to do such reductions.

Program Repair and Test-Input Generation. Due to the pressing demand for reliable software, automatic program repair has steadily gained research interests and produced many novel repair techniques. *Synthesis* repair approaches, e.g., AFix [23], Angelix [31], SemFix [33], FoRenSiC [8], Gopinath et al. [19], Jobstmann et al. [24], generate constraints and solve them for patches that are correct by construction (i.e., guaranteed to adhere to a specification or pass a test suite). In contrast, *generate-and-validate* repair approaches, e.g., GenProg [48], Pachika [12], PAR [26], Debroy and Wong [13], Prophet [30], find multiple repair candidates (e.g., using stochastic search or invariant inferences) and verifies them against given specifications.

The field of test-input generation has produced many practical techniques and tools to generate high coverage test data for complex software, e.g., fuzz testing [32, 16], symbolic execution [9, 10], concolic (combination of static and dynamic analyses) execution [17, 39], and software model checking [7, 6]. Companies and industrial research labs such as Microsoft, NASA, IBM, and Fujitsu have also developed test-input generation tools to test their own products [18, 3, 4, 29]. Our work allows program repair and synthesis approaches directly apply these techniques and tools.

6 Conclusion

We constructively prove that the template-based program synthesis problem and the reachability problem in program verification are equivalent. This equivalence connects the two problems and enables the application of ideas, optimizations, and tools developed for one problem to the other. To demonstrate this, we develop CETI, a tool for automated program repair using test-input generation techniques that solve reachability problems. CETI transforms the task of synthesizing program repairs to a reachability problem, where the results produced by a test-input generation tool correspond to a patch that repairs the original program. Experimental case studies suggest that CETI has higher success rates than many other standard repair approaches.

References

1. R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. *Dependable Software Systems Engineering*, 40:1–25, 2015.
2. R. Alur, P. Cerný, and A. Radhakrishna. Synthesis through unification. In *CAV*, volume 9207, pages 163–179. Springer, 2015.
3. S. Anand, C. S. Păsăreanu, and W. Visser. JPF–SE: A symbolic execution extension to Java Pathfinder. In *TACAS*, pages 134–138. Springer, 2007.
4. S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *ISSTA*, pages 261–272. ACM, 2008.
5. P. Attie, A. Cherri, K. D. Al Bab, M. Sakr, and J. Saklawi. Model and program repair via sat solving. In *MEMOCODE*, pages 148–157. IEEE, 2015.
6. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, pages 1–3. ACM, 2002.
7. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *Soft. Tools for Technol. Transfer*, 9(5-6):505–525, 2007.
8. R. Bloem, R. Drechsler, G. Fey, A. Finder, G. Hofferek, R. Könighofer, J. Raik, U. Repinski, and A. Sülflow. FoREnSiC–an automatic debugging environment for C programs. In *HVC*, pages 260–265. Springer, 2013.
9. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224. USENIX Association, 2008.
10. C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.
11. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.
12. V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *ASE*, pages 550–554. IEEE, 2009.
13. V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation*, pages 65–74. IEEE, 2010.
14. H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.
15. ExCAPE: Expeditions in computer augmented program engineering. http://excape.cis.upenn.edu, 2016-10-19.
16. J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *USENIX Windows System Symposium*, pages 59–68, 2000.
17. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *PLDI*, 40(6):213–223, 2005.
18. P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium*, pages 151–166, 2008.
19. D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using SAT. In *TACAS*, pages 173–188. Springer, 2011.
20. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330. ACM, 2011.
21. S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, Aug. 2012.

22. J. Jeon, X. Qiu, J. Fetter-Degges, J. S. Foster, and A. Solar-Lezama. Synthesizing framework models for symbolic execution. In *ICSE*, pages 156–167. ACM, 2016.
23. G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, pages 389–400. ACM, 2011.
24. B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, pages 226–238. Springer, 2005.
25. J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ICSE*, pages 273–282. IEEE, 2005.
26. D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE*, pages 802–811. ACM, 2013.
27. R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In *FMCAD*. IEEE, 2011.
28. R. Könighofer and R. Bloem. Repair with on-the-fly program analysis. In *HVC*, pages 56–71. Springer, 2013.
29. G. Li, I. Ghosh, and S. P. Rajan. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In *CAV*, pages 609–615. Springer, 2011.
30. F. Long and M. Rinard. Automatic patch generation by learning correct code. In *POPL*, volume 51, pages 298–312. ACM, 2016.
31. S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE*, pages 691–701. ACM, 2016.
32. B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.
33. H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *ICSE*, pages 772–781. ACM, 2013.
34. T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using Dynamic Analysis to Discover Polynomial and Array Invariants. In *ICSE*, pages 683–693. IEEE, 2012.
35. T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Connecting program synthesis and reachability. Technical report, University of Nebraska, Lincoln, Oct 2016.
36. G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
37. H. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. of the American Mathematical Society*, 74(2):358–366, 1953.
38. S. Saha, P. Garg, and P. Madhusudan. Alchemist: Learning guarded affine functions. In *CAV*, pages 440–446, 2015.
39. K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423. Springer, 2006.
40. R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26. ACM, 2013.
41. A. Solar-Lezama. *Program synthesis by sketching*. PhD thesis, University of California, Berkeley, 2008.
42. A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *PLDI*, pages 167–178. ACM, 2007.
43. A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. *PLDI*, 40:281–294, 2005.
44. S. Srivastava. *Satisfiability-based program reasoning and program synthesis*. PhD thesis, University of Maryland, 2010.
45. S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326. ACM, 2010.
46. S. Srivastava, S. Gulwani, and J. S. Foster. Template-based program verification and program synthesis. *Soft. Tools for Technol. Transfer*, 15(5-6):497–518, 2013.

47. SyGuS: Syntax-guided synthesis competition. www.sygus.org, 2016-10-19.
48. W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically Finding Patches Using Genetic Programming. In *ICSE*, pages 364–367. IEEE, 2009.

7 Appendix: Correctness Proofs

7.1 Formal Semantics

We use standard operational semantics to reason about the meanings and executions of programs [36]. A state σ maps variables to values. Given a program P, the *large-step* judgment $\langle P, \sigma \rangle \Downarrow \sigma'$ means that P, when executed in an initial state σ, terminates in state σ'. The *small-step* judgment $\langle P, \sigma \rangle \rightarrow \langle P', \sigma' \rangle$ means that if P is executed in state σ for one single step, it transitions to (or reduces to) program P' in state σ'. We write $\rightarrow^*$ for the reflexive, transitive closure of that relation (i.e., zero or more steps). Both large-step and small-step judgments are proved or constructed using *derivations*, tree-structured mathematical objects made of well-formed instances of inference rules. We write $D :: \ldots$ to indicate that D is a derivation proving a judgment. There is often only one inference rule for each statement type (e.g., one assignment rule for reasoning about assignment statements), which admits reasoning by *inversion*: for example, if $D :: \langle P, \sigma \rangle \Downarrow \sigma'$ and P is an assignment statement, then D must be the assignment rule (and not, for example, the while loop rule), and vice-versa.

We write $\sigma \models B$ to denote that a state σ makes a predicate B true. We write $wp(P, B)$ for the weakest precondition of P with respect to postcondition B. If $A = wp(P, B)$, then if P executes in a state that makes the precondition A true and it terminates, then it terminates in a state making postcondition B true.

We use large-step operational semantics when reasoning about synthesis, where we focus on the final value of the program (i.e., its behavior on a test). We use small-step operational semantics when reasoning about reachability, where intermediate steps matter (i.e., did the program execution visit a particular label?). We use induction on the structure of a derivation to show that a property holds for all executions of all programs (informally, if the property holds for one-step programs like `skip`, and it also holds whenever one more execution step is added, then it always holds). We use weakest preconditions to reason about special conditional statements that encode test cases.

Our use of small-step and large-step semantics, as well as structural induction on derivations and preconditions, is standard (i.e., the proofs and the constructions are novel, but not the proof machinery). The reader is referred to Plotkin [36] for a thorough introduction.

7.2 Reducing Synthesis to Reachability

We prove correct the constructive reduction given in Section 3.2. The correctness of $\mathsf{GadgetS2R}$ hinges on two key invariants. First, function calls in the derived program P have the same behavior as template functions in the original program Q. Second, location L is reachable if and only if values c_i can be assigned to variables v_i such that Q passes all of the tests. Formally, we say that $\mathsf{GadgetS2R}$ maintains the invariants described by the following Lemmas. For brevity, we write $\sigma_1 \simeq_V \sigma_2$ to denote $\forall x.\ x \notin V \Rightarrow \sigma_1(x) = \sigma_2(x)$ — that is, state equivalence modulo a set of variables V.

We first show that p_q(input) behaves as $q[c_1, \dots, c_n]$(input), when the new v_i variables in P are assigned the values c_i. Formally,

Lemma 1. *Let* $(P, L) = \mathsf{GadgetS2R}(Q, S, T)$*. For all states* $\sigma_1, \sigma_2, \sigma_3$*, all values* $c_1, \dots, c_n, i$*, and all functions* $q \in Q$*, if* $\sigma_1(v_i) = c_i$ *then* $D_1 :: \langle p_q(i), \sigma_1 \rangle \Downarrow \sigma_2$ *if both* $D_2 :: \langle q[c_1, \dots, c_n](i), \sigma_1 \rangle \Downarrow \sigma_3$ *and* $\sigma_2 \simeq_{\{v_1, \dots, v_n\}} \sigma_3$*.*

Proof. The proof proceeds by induction on the structure of the operational semantics derivation of D_1. Let σ_1 be arbitrary with $\sigma_1(v_i) = c_i$.

Note that by Step 2 of the GadgetS2R construction, each $p_q(i) \in P$ corresponds to $q[c_1, \dots, c_n](i) \in Q$ in a particular manner. Indeed, all subexpressions in p_q and q are identical *except* for one case. References to template parameters $\boxed{c_i}$ in the template program Q correspond exactly to references to variables v_i in the derived reachability program P.

Thus, by inversion, the structure of D_1 corresponds exactly to the structure of any D_2 except for variable references. The inductive proof considers all of the cases for the derivation D_1. For brevity, we show only the non-trivial case.

Case 1 (Template Variable Read). Suppose D_1 is:

$$\frac{\sigma_2 = \sigma_1[a \mapsto \sigma_1(v_i)]}{\langle a := v_i, \sigma_1 \rangle \Downarrow \sigma_2}\ \textsf{assign-variable}$$

By inversion and the construction of P, D_2 is:

$$\frac{\sigma_3 = \sigma_1[a \mapsto c_i]}{\langle a := \boxed{c_i}, \sigma_1 \rangle \Downarrow \sigma_3}\ \textsf{assign-template-parameter}$$

Note that in D_1, v_i is a normal program variable expression (referring to a variable introduced in Step 1 of the construction), while in in D_2, $\boxed{c_i}$ is a reference to a template parameter. We must show that $\sigma_2 \simeq_{\{v_1, \dots, v_n\}} \sigma_3$. Since σ_2 and σ_3 agree with σ_1 on all variables except a, it only remains to show that $\sigma_2(a) = \sigma_3(a)$ (since $a \notin \{v_1, \dots, v_n\}$). By the σ definitions from D_1 and D_2, that obligation simplifies to showing that $\sigma_1(v_i) = c_i$, which was part of the input formulation for this Lemma. (Intuitively, $\sigma_1(v_i) = c_i$ means that the reachability analysis assigned the values c_i to each variable v_i.)

The other cases, which do not involve the template parameters v_i, are direct (the inference rule used in D_1 will exactly mirror the inference rule used in D_2).

Having established that the executions of P's functions mirror Q's functions (modulo the template parameters, which are held constant), we now establish that reaching L in P via assigning each v_i the value c_i corresponds to $Q[c_1, \dots, c_n]$ passing all of the tests.

Lemma 2. *Let* $(P, L) = \mathsf{GadgetS2R}(Q, S, T)$*. The execution of* P *reaches* L *starting from state* σ_1 *if and only if* $\sigma_1 \models wp(Q[c_1, \dots, c_n](i_1), \text{result} = o_1) \wedge \dots \wedge wp(Q[c_1, \dots, c_n](i_m),$
$\text{result} = o_m)$*, where* $\sigma_1(v_i) = c_i$*,* result *denotes the return value of* Q*, and* wp *is the weakest precondition.*

Proof. By construction, there is exactly one location L in P: "if e: [L]" is the body of $p_{\text{gadget}} \in P$. From Step 3, e has the general form $f(i_1) = o_1 \wedge \cdots \wedge f(i_m) = o_m$. By the standard weakest precondition definitions for if, conjunction, equality and function calls, we have that L is reachable if and only if $\sigma_1 \models wp(p_{\text{gadget}}(i_i), \text{result} = o_1) \wedge \cdots \wedge wp(p_{\text{gadget}}(i_m), \text{result} = o_m)$. That is, the label is reachable iff the derived gadget program passes the tests (with variables $v_i \mapsto c_i$). We now show that this occurs if and only if the original template program passes the tests (with template parameters $\boxed{c_i} \mapsto c_i$).

By conjunction elimination, for each test (i, o) we have $\sigma_1 \models wp(p_{\text{gadget}}(i), \text{result} = o)$. By the soundness and completeness of weakest preconditions with respect to operational semantics,[9] we have that $\langle p_{\text{gadget}}(i), \sigma_1 \rangle \Downarrow \sigma_2$ iff $\sigma_2 \models \text{result} = o$ (equivalently, $\sigma_2(\text{result}) = o$). By Lemma 1, we have $\langle Q[c_1, \ldots, c_n](i), \sigma_1 \rangle \Downarrow \sigma_3$ with state $\sigma_2 \simeq_{\{v_1,\ldots,v_n\}} \sigma_3$. Since result $\notin \{v_1, \ldots, v_n\}$, we have $\sigma_3(\text{result}) = \sigma_2(\text{result}) = o$. That is, the template program Q, filled in with the concrete values c_i, when run on any test input i yields the expected result o. This occurs if and only if an execution of P reaches L starting in a state that maps each v_i to c_i.

This Lemma leads directly to the main result for this direction of the reduction.

Theorem 3. *Let Q be a program with template parameters $S = \{\boxed{c_1}, \ldots, \boxed{c_n}\}$ and a test suite $T = \{(i_1, o_1), \ldots\}$. Let $(P, L) = \mathsf{GadgetS2R}(Q, S, T)$. Then there exist parameter values c_i such that $\forall (i, o) \in T \,.\, (Q[c_1, \ldots, c_n])(i) = o$ if and only if there exist input values t_i such that the execution of P with $v_i \mapsto t_i$ reaches L. That is, the template-based synthesis problem in Definition 1 is reducible to the reachability problem in Definition 2.*

Proof. This follows directly from Lemma 2 with t_i equal to c_i.

7.3 Reducing Reachability to Synthesis

We now address the correctness of the constructive reduction given in Section 3.3. The correctness of GadgetR2S also relies on two key invariants. First, for any c_i, execution in the derived templated program Q with $\boxed{c_i} \mapsto c_i$ mirrors execution in P with $v_i \mapsto c_i$ up to the point when L is reached (if ever). Second, the exception `REACHED` is raised in Q iff location L is reachable in P.

We first show that $q_p[c_1, \ldots, c_n]$ behaves as p with $v_i \mapsto c_i$. Because our construction uses exceptions, we phrase the lemma formally using small-step operational semantics.

[9] Our approach is thus sound and relatively complete in practice. An implementation would use provability ($\vdash$), such as from an SMT solver, instead of truth ($\models$). While sometimes sound on restricted domains, decision procedures are not complete in general, in which case a failure to solve the reachability problem P does *not* imply that the synthesis problem Q has no solution.

Lemma 3. *Let $Q, S, T = \mathsf{GadgetR2S}(P, L, v_i)$. For all states $\sigma_1, \sigma_2, \sigma_3$, all values c_i, all functions $p \in P$, and all programs p' and q'_p, if $\sigma_1(v_i) = c_i$ then we have $D_1 :: \langle p, \sigma_1 \rangle \to^* \langle p', \sigma_2 \rangle$ with L not executed in D_1 iff $D_2 :: \langle q_p, \sigma_1 \rangle \to^* \langle q'_p, \sigma_3 \rangle$ with* `raise REACHED` *not executed in D_2 and $\sigma_2 \simeq_{v_i} \sigma_3$ where q'_p is equal to p' with each v_i replaced by $\boxed{c_i}$ and each $[L]$ replaced by* `raise REACHED`*.*

Proof. The proof proceeds by induction on the structure of the operational semantics derivation D_1. The proof follows the reasonings shown in Section 3.2 and is elided in the interest of brevity.

Having established that the execution of Q's functions mirrors the execution of P's functions before the location L is reached (when $\boxed{c_i} = v_i = c_i$), we now demonstrate that exception `REACHED` is raised in Q iff location L is reachable in P:

Lemma 4. *Let $Q, S, T = \mathsf{GadgetR2S}(P, L, v_i)$. For all values c_i, Then $Q[c_1, \ldots, c_n]() = 1$ iff location L is reachable in P starting from a state with $v_i \mapsto c_i$.*

Proof. The proof follows from the instantiation of Lemma 3. By construction and the operational semantics rules for *try-catch*, q_{main} returns 1 iff q_p did raise `REACHED`. By construction, there is exactly one occurrence of `raise REACHED` in Q, and that statement occurs at a point corresponding to the singular location L in P. Note that q_p is derived from p and shares the same structure except that reads from variables v_i in P are replaced with reads from template variables $\boxed{c_i}$ in Q and the location L is replaced by `raise REACHED`.

Thus we apply Lemma 3 with $p = p_{main}$, $q_p = q_{main}$ and σ_1 such that $\sigma_1(v_i) = c_i$ (from the statement of Lemma 4). Since Lemma 3 applies to all program points p' and q'_p up to the first execution of L (resp. `raise REACHED`), we have a derivation D_1 starting in σ_1 and ending in $p' = [L]; p''$ iff we have a matching derivation D_2 starting in σ_1 and ending in $q'_p = \texttt{raise REACHED}; q''_p$.

Thus, L is reachable in P starting from a state $\sigma_1(v_i) = c_i$ iff $Q[c_1, \ldots, c_n]()$ returns 1.

This Lemma leads directly to the Theorem for this direction of the reduction.

Theorem 4. *Let P be a program with a location L and global variables $v_1, \ldots, v_n$. Let $Q, S, T = \mathsf{GadgetR2S}(P, L, v_i)$. Then there exist input values t_i such that the execution of P with $v_i \mapsto t_i$ reaches L iff there exist parameter values c_i such that $\forall (i, o) \in T.(Q[c_1, \ldots, c_n])(i) = o$. That is, the reachability problem in Definition 2 is reducible to the template-based synthesis problem in Definition 1.*

Proof. This follows from Lemma 4 with t_i equal to c_i. Note that T is the singleton set containing $(\emptyset, 1)$ by construction, so the universal qualifier reduces to the single assertion $Q[c_1, \ldots, c_n]() = 1$.