Lecture Notes: Program Repair as Reachability

17-355/17-665/17-819: Program Analysis (Spring 2020)
Claire Le Goues*
clegoues@cs.cmu.edu

There are useful correspondences between techniques we have used for verification and those we then explored for synthesis and repair. For example, verification condition generation began as a method to prove programs correct; generating these conditions forwards rather than backwards allowed us to develop a way to generalize testing through symbolic execution; we could then use symbolic/concolic execution as a way to perform synthesis for repair of certain types of defects.

Today, we will explore a formal connection between program synthesis (for repair, specifically) and verification, formulated as a *reachability* problem.

1 Template-based program synthesis for repair

One general way to formulate program repair is as a problem of selecting and appropriately instantiating one or more *repair templates* at the appropriate points in a program. We define a general syntax for a templated program along the following lines (borrowing from the WHILE syntax, but simplifying somewhat for the purposes of this discussion):

$$
\begin{array}{rcll}
S & ::= & x := a & \\
 & | & \text{skip} & \\
 & | & S_1;\ S_2 & \\
 & | & \text{if } P \text{ then } S_1 \text{ else } S_2 & \\
 & | & \text{while } P \text{ do } S & \\
\end{array}
\qquad
\begin{array}{rcll}
a & ::= & a_1 + a_2 & \\
 & | & a_1 - a_2 & \\
 & | & \boxed{c_i} & \textit{called a template parameter!} \\
 & | & \ldots & \\
 & & & \\
\end{array}
$$

Given a templated program with template parameters $c_1 \ldots c_n$ and given template values $\bar{v} = v_1 \ldots v_n$ (corresponding to expressions or constants), we can *instantiate* thatt emplate on those values to yield a non-templated program. We can define instantiation in a straightforward, syntax-directed way:

$$
\begin{array}{lcl}
inst(\text{skip}, \bar{v}) & \rightarrow & \text{skip} \\
inst(S_1; S_2, \bar{v}) & \rightarrow & inst(S_1, \bar{v}); inst(S_2, \bar{v}) \\
inst(x := a, \bar{v}) & \rightarrow & x = inst(a, \bar{v}) \\
inst(\boxed{c_i}, \bar{v}) & \rightarrow & v_i
\end{array}
$$

The *template-based program synthesis problem* is then defined as follows:

*These notes draw from Nguyen et al., "Connecting Program Synthesis and Reachability: Automatic Program Repair using Test-Input Generation", TACAS 2017 and a set of course materials generously provided by Wes Weimer.

Given a templated program P with template parameters $c_1 \ldots c_n$, and a set T of input-output pairs (tests), do there exist template values $\bar{v} = v_1 \ldots v_n$ such that for all $\langle \alpha_0, \beta_0 \rangle \ldots \langle \alpha_n, \beta_n \rangle$ in T, $(inst(P, \bar{v}))(\alpha_i) = \beta_i$?

Note that we are ambivalent as to the mechanism used to identify $\bar{v}$, and that many of the inductive techniques we have discussed either for synthesis proper or for program repair specifically fit in this framing (consider syntax-guided synthesis).

We can extend this representation to program repair by constructing templated programs from the original program and replacing potentially buggy lines with potential template $\boxed{c_i}$. By synthesizing some code to fill arbitrary hole, the repair effectively becomes "delete buggy statement X and replace with instantiated template Y." Overall, these templates focus on expression-level manipulation; extending this framing to statement-level modifications is an open problem.

For the purposes of this exposition, we ignore fault localization, focus on single-edit repairs, and on templates encoding linear combinations of variables. Actual repair based on these ideas generalizes by trying several locations/templates in some order. More complicated (e.g., non-linear) templates are usable as well.

2 Program Reachability

The problem of reachability as applied to programs asks, very generally, whether given a program P, a set of program variables $x_1 \ldots x_n$ and some program label L, do there exist values $c_1 ... c_n$ such that P with $x_i = c_i$ reaches label L in finite time?

We have seen this applied to finding bugs using symbolic execution (e.g., formulating buffer overflows as reaching an error state via program transformation). Test generation can also be viewed as generating c_i for test inputs with L corresponding to the end of a desired execution path. It is also applicable to model checking, as we will see in future course lessons.

The following code example revisits the idea/intuition, calling back to our prior discussions on test generation:

```
1 int x, y; /* global input */
2 int P() {
3 if (2 * x == y) {
4   if (x > y + 10)
5       [ L ]
6   return 0;
7 }
```

Here, $x = -20, y = -40$ reaches the label.

3 Reducing Synthesis To Reachability

Both reachability and synthesis are undecidable in general. You may recall how reductions work from prior theory or algorithms courses. In brief, Problem A is reducible to Problem B if an efficient algorithm for B could be used as a subroutine to solve A efficiently. A *gadget* is a subset of a problem instance that simulates the behavior of one of the fundamental units of a different problem.

3.1 Defining GadgetS2R

Thus, given an instance of a synthesis (repair) problem, and assuming we have an oracle that can solve reachability, let us convert the synthesis instance into a reachability instance. If we can do this efficiently, any existing reachability tool/technique (e.g., one that performs symbolic or concolic execution) could be used to repair programs.

Give Q, a template program with a set of template parameters $S = \{c_1, \ldots, c_n\}$ and a set of finite tests $T = (\alpha_1, \beta_1), \ldots$, construct $GadgetS2R(Q, S, T)$ which returns a new program P with a special location L, as follows:

1. For every template parameter $\bar{c_i}$, add a fresh global variable v_i. A solution to the reachability instance is an assignment of concrete values c_i to 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 $\bar{c_i}$ 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:

   $$e = \bigwedge_{(\alpha_i, \beta_i) \in T} main_{QP}(\alpha_i) = \beta_i$$

   where $main_{QP}$ is a function in P corresponding to the starting function $main_Q$ in Q. The body of $main_P$ is then a single conditional statement that leads to a fresh target location L iff e is true.

4. P overall consists of the declaration of new variables, the functions q_P, and the starting function $main_P$

3.2 Illustrative example

Consider the code sample we used in discussing semantics-based program repair:

```
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
8    if (bias > down)
9      r = 1;
10   else
11     r = 0;
12   return r
13   }
```

And a set of test cases that highlight the bug:

Test	Inputs			Output		
	in	up	down	expected	observed	Passed?
1	1	0	100	0	0	Yes
2	1	11	110	1	0	No
3	0	100	50	1	1	Yes
4	1	-20	60	1	0	No
5	0	0	-10	1	1	Yes

Assuming we select the buggy line and attempt a linear combination of variables as a template, this transforms the buggy code into a templated program as so:

```
1  int is_upward(int in, int up, int down) {
2    int bias, r;
3    if (in)
4      bias = c0 + c1 * bias + c2 * in + c3 * up + c4 * down;
5    else
6      bias = up;
7
8    if (bias > down)
9      r = 1;
10   else
11     r = 0;
12   return r;
13   }
```

The reachability instance corresponding to this program then looks like:

```
1  int c0, c1, c2, c3, c4; /* global input */
2
3  int P_is_upward(int in, int up, int down) {
4    int bias, r;
5    if (in)
6      bias = c0 + c1 * bias + c2 * in + c3 * up + c4 * down;
7    else
8      bias = up;
9
10   if (bias > down)
11     r = 1;
12   else
13     r = 0;
14   return r;
15 }
16 int main () {
17   if (p_is_upward(1,0,100) == 0 &&
18       p_is_upward(1,11,110) == 1 &&
19       p_is_upward(0,100,50) == 1 &&
20       p_is_upward(1,-20,60) == 1 &&
21       p_is_upward(0,0,10) == 0 &&
22       p_is_upward(0,0,-10) == 1) {
23    [L] // label!
24   }
25   return 0;
26 }
```

A valid solution to the reachability problem identifies global inputs `c0 = 100, c1 = 0, c2 = 0, c3 = 1, c4 = 0`; this corresponds to instantiating the template on line 4 with `bias = up + 100;`, a valid patch for this bug.

4 Proof of correctness

To prove the correctness of this reduction, we must show that the constructed reachability instance is solvable (with values $c_1 \ldots c_n$) iff the original synthesis instance is solvable (with those same values). The reachability instance is solved if those values cause execution to reach L, while the synthesis instance is solved if those same values cause every test to pass.

The proof uses standard operational semantics to reason about the meaning and executions of programs. We use large-step semantics ($\Downarrow$) to reason about synthesis, which focuses on the final value of the program (its behavior on a test). W uses small-step semantics ($\rightarrow$) to reason about reachability, where the intermediate steps matter (was a particular label reached?). The proof uses induction on the structure of a derivation to show that a property holds for all exeuctions of all programs, and weakest preconditions to reason about the special conditional statements that encode test cases.

The high-level proof structure is as follows:

- Lemma 1: The reachability instance method and the synthesis instance method agree on the values of all (non-template) variables.
- Lemma 2: If the reachability instance reaches L from a state S (with values c_1 $ldotsc_n$), then that state and values model the weakest precondition of the synthesis instance method passing each test.
- Theorem 1. The synthesis instance is solvable iff the reachability instance is solvable (with the same values).

4.1 Lemma 1: Agree on Vars

The idea here is to show that the derived program ($p_q(\alpha_i)$) behaves the same as the original program $q[c_0, \ldots, c_n](\alpha_i)$[1] when the new variables v_i in P are assigned the values c_i.

Formally, let Q be the input synthesis instance method with template variables $v_1 \ldots v_n$. Let $P = GadgetS2R(Q)$ be the reachability instance corresponding to method P. For all states E_1, E_2, E_3, all values $c_1, \ldots c_n$, all input values x, we seek to prove that it holds that:

> if $E_1(v_i) = c_i$, then $D_1 :: \langle P(x), E_1 \rangle \Downarrow E_2$ iff
> $D_2 :: \langle inst(Q, \bar{c}), E_1 \rangle \Downarrow E_3$
> and $\forall y \neq v_i, E_2(y) = E_3(y)$.

The proof proceeds by induction on the structure of an operational semantics derivation D_1. Let E_1 be arbitrary, except with $E_1(v_i) = c_i$. Importantly, every $p_q(i) \in P$ corresponds to $q[c_1, \ldots, c_n](e) \in Q$, in the sense that all subexpressions in p_q and q are identical *except* that references to template parameters $\boxed{c_i}$ in Q correspond to references to variables v_i in P. Thus, by inversion, the structure of D_1 corresponds exactly to the structure of D_2, except for the variable references. Thus, although for completeness we'd consider all cases for the derivation D_1, in practice we'll only show the interesting case, which is a read/assignment of a templated variable.

Case: Suppose D_1 (reachability instance) is:

[1] $q[c_0, \ldots, c_n]$ is another way of saying "q instantiated on $c_0, \ldots, c_n$"

$$\frac{E_2 = E_1[a \mapsto E_1(v_i)]}{\langle a := v_i, E_1 \rangle \Downarrow E_2} \textit{ assign}$$

By inversion and the construction of P, D_2 is:

$$\frac{E_3 = E_1[a \mapsto c_i]}{\langle a := exp, E_1 \rangle \Downarrow E_3} \textit{ assign}$$

where $exp = inst(\boxed{c_i}, \bar{c}) = c_i$

Now, we know that $E_2 = E_1[a \rightarrow E_1(v_i)]$ and $E_3 = E_1[a \rightarrow c_i]$. We know that E_2 and E_3 agree on all variables except a (because they are the same as E_1, for all variables except a). So, this obligation simplifies to showing that $E_1(v_i) = c_i$. This is actually one of the assumptions in the statement of the lemma. (Intuitively, it means the reachability analysis assigned c_i to each variable v_i to reach the label L.)

The other cases are direct, because they do not involve the template paramters v_i: the inference rule used in D_1 will exactly mirror the inference rule used in D_2.

We have therefore established that the executions of P's functions mirror Q's functions, modulo the template parameters, which are held constant.

4.2 Lemma 2 (Reach L = Pass tests)

We now establish that reaching L in P by assigning each v_i the value c_i corresponds to $Q[c_i, \ldots, c_n]$ passing all of the tests.

Let Q be the input synthesis instance method with template variables $v_1 \ldots . v_n$ and tests $\langle \alpha_1, \beta_n \rangle$ Let P = GadgetS2R(Q) be the reachability instance method `main`.

We know that the execution of P reaches L starting from state E_1 iff $E_1 \vDash wp(inst(Q, \bar{c})(\alpha_1), result = \beta_1) \wedge \ldots wp(inst(Q, \bar{c}(\alpha_n), result = \beta_n)$ where $E_1(v_i) = c_i$. By gadget construction, we also know there is only one label L in P, `if` e `then` $[L]$, and that e is of the form $f(\alpha_1) = \beta_1 \wedge \ldots f(\alpha_n) = \beta_n$.

By standard weakest precondition definitions for if, conjunction, equality and function calls, we have that L is reachable iff $E_1 \vDash wp(result := f(\alpha_1), result = \beta_1) \wedge \ldots wp(result := f(\alpha_n), result = \beta_n)$. What we *want* is that L is reachable iff $E_1 \vDash wp(result := inst(Q, \bar{c})(\alpha_1), result = \beta_1) \wedge \ldots wp(result := inst(Q, c)(\alpha_n), result = \beta_n)$

So, we have to show that $E_1 \vDash wp(result := f(\alpha_i), result = \beta_i)$ iff $E_1 \vDash wp(result := inst(Q, \bar{c})(\alpha_i), result = \beta_i)$. f here is the method from GadgetS2R(Q).

By the soundness and completeness of weakest preconditions wrt operational semantics, we have $\langle result := f(\alpha_i), E_1 \rangle \Downarrow E_2$ iff $E_2 \vDash result = \beta_i$. By Lemma 1, we have $\langle result := inst(Q, \bar{c})(\alpha_i), E_1 \rangle \Downarrow E_3$ iff $E_1(y) = E_3(y)$ for all $y \neq v_i$. Since "result" $\neq v_i$, $E_1(result) = E_3(result)$ (we know this from lemma 1) and $E_3(result) = \beta_i$ (because of the weakest precondition reasoning above). So, running the template program Q instantiated with $c_i = v_i$ on a test input produces the required output.

4.3 Correctness

This leads us to the correctness theorem, which is as follows: Let Q be the input synthesis instance method with template variables $v_1 ... v_n$ and tests $\langle \alpha_1, \beta_n \rangle$. Let $P = GadgetS2R(Q)$ be the reachability instance method `main`.

There exist parameter values c_i such that for all $\langle \alpha_i, \beta_i \rangle, inst(Q, \bar{c})(\alpha_i) = \beta_i$ iff there exists input values t_i s.t. the execution of P with $v_i \rightarrow t_i$ reaches L. The proof is from Lemma 2 with $t_i = c_i$.

Note that we can also carry out a constructive reduction going in the other direction. That is, suppose we are given an instance of program reachability. *Can we convert it into a program synthesis instance to solve it?*[2]

5 Implications

Program reachability tools and techniques are much more mature than program repair tools. This correspondence between the problems therefore suggests a way to use reachability to attempt to fix bugs in programs. It proceeds by, for every potentially buggy line, in some ranked order, and then for every possible considered repair template, also in some ranked order, converting the repair instance to a reachability instance and then calling an off-the-shelf reachability tool (e.g., an SMT solver-based concolic execution engine like KLEE). If the label is reachable, the discovered parameters in the satisfying model can be returned for instantiation as a program patch.

Overall, this correspondance helps illustrate how all of the techniques we have discussed (e.g., test generation, model checking, verification) are seeking, in some way, to statically reason about dynamic execution. We can sometimes take advantage of this correspondence to discover new techniques, as this particular reduction demonstrates.

[2]We elide these details but note they are expanded upon in the original publication.