Connecting Program Synthesis and Reachability:

Automatic Program Repair using Test-Input Generation

ThanhVu (Vu) Nguyen*,
Westley Weimer†, Deepak Kapur‡, Stephanie Forrest‡

*University of Nebraska, †University of Michigan, ‡University of New Mexico

TACAS 2017

Connecting Verification to Synthesis

Program Verification

Checks if program satisfies a given spec

Significant research development, e.g.,
formal methods, software testing

Connecting Verification to Synthesis

Program Verification

Checks if program satisfies a given spec


Significant research development, e.g., formal methods, software testing

Program Synthesis

Creates program that meets a given spec


Less work compared to verification, *"among the last tasks that computers will do well"*

Connecting Verification to Synthesis

Program Verification

Checks if program satisfies a given spec


Significant research development, e.g., formal methods, software testing

Program Synthesis

Creates program that meets a given spec


Less work compared to verification, *"among the last tasks that computers will do well"*

- **Implicit assumption**: verification and synthesis are related
  - Many verification techniques adopted to synthesize programs
  - Synthesis using constraint solving, program repairs using symexe, etc
- **Goal**: finding and formalizing this relation
  - Allow for comparisons between complexity and underlying structures
  - Leverage existing verification/synthesis techniques and tools to synthesize/check programs

Contributions

Relation between certain formulations of synthesis and verification

- Focus on *template-based synthesis* and view verification as *reachability* task
- Constructively prove that they are *equivalent*
  1. Reduce template-based synthesis into a program consisting of a special loc, reachable only when code could be synthesized
  2. Transform a reachability problem into a specific template-based synthesis instance, solvable only when the loc in the original problem is reachable
- Thus, reachability solvers can be used to generate code, and conversely, synthesis tools can be applied to check reachability

Contributions

Relation between certain formulations of synthesis and verification

- Focus on *template-based synthesis* and view verification as *reachability* task
- Constructively prove that they are *equivalent*
  1. Reduce template-based synthesis into a program consisting of a special loc, reachable only when code could be synthesized
  2. Transform a reachability problem into a specific template-based synthesis instance, solvable only when the loc in the original problem is reachable
- Thus, reachability solvers can be used to generate code, and conversely, synthesis tools can be applied to check reachability

New automatic program repair technique

- Treat program repair as a form of template-based synthesis
- CETI: repair C programs violating *test-suite specification*
  - Use fault localization to rank suspicious stmts
  - Transform program, suspicious stmts, and test suite into reachability prob
  - Apply off-the-shelf test-input generation tool to find inputs
  - Map input values to changes allowing program to pass test suite

Example: Synthesize Program Repairs

```
int isUp(int in,int up,int down){
  int bias, r;
  if (in)
    //fix: bias = up + 100
    bias = down;
  else
    bias = up;
  if (bias > down)
    r = 1;
  else
    r = 0;
  return r;
}
```

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	✓

Example: Synthesize Program Repairs

```
int isUp(int in,int up,int down){
  int bias, r;
  if (in)
    //fix: bias = up + 100
    bias = down;
  else
    bias = up;
  if (bias > down)
    r = 1;
  else
    r = 0;
  return r;
}
```

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	✓

- Synthesize programs using *templates*,
  e.g., linear comb of variables: $\boxed{c_0} + \boxed{c_1} v_1 + \boxed{c_2} v_2$

Example: Synthesize Program Repairs

```
int isUp(int in,int up,int down){
  int bias, r;
  if (in)
    //fix: bias = up + 100
    bias = down;
  else
    bias = up;
  if (bias > down)
    r = 1;
  else
    r = 0;
  return r;
}
```

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	✓

- Synthesize programs using *templates*, e.g., linear comb of variables: $\boxed{c_0} + \boxed{c_1} v_1 + \boxed{c_2} v_2$
- Replace suspicious stmt bias = down; with bias $= \boxed{c_0} + \boxed{c_1}$bias $+ \boxed{c_2}$in $+ \boxed{c_3}$up $+ \boxed{c_4}$down;

Example: Synthesize Program Repairs

```
int isUp(int in,int up,int down){
  int bias, r;
  if (in)
    //fix: bias = up + 100
    bias = down;
  else
    bias = up;
  if (bias > down)
    r = 1;
  else
    r = 0;
  return r;
}
```

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	✓

- Synthesize programs using *templates*, e.g., linear comb of variables: $\boxed{c_0} + \boxed{c_1} v_1 + \boxed{c_2} v_2$
- Replace suspicious stmt bias = down; with bias $= \boxed{c_0} + \boxed{c_1}$bias $+ \boxed{c_2}$in $+ \boxed{c_3}$up $+ \boxed{c_4}$down;
- Find unknowns $\boxed{c_i}$ by creating and solving a special *reachability* instance

Example: Constructing Reachability Instance

```
int isUp(int in,int up,int down){
  int bias, r;
  if (in)
    //template stmt
    bias = c0+c1bias+c2in+c3up+c4down;
  else
    bias = up;
  if (bias > down)
    r = 1;
  else
    r = 0;
  return r;
}
```

Test	Inputs: in	up	down	Output: expected	observed	Passed?
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	✓

Transform the program, test suite, and template stmts into a program Q having:

1. the template code with the params $\boxed{c_i}$ represented as *global vars*
2. a location L guarded by conditional expressions representing test cases

Example: Constructing Reachability Instance

```
int isUp(int in,int up,int down){
  int bias, r;
  if (in)
    //template stmt
    bias = c0+c1bias+c2in+c3up+c4down;
  else
    bias = up;
  if (bias > down)
    r = 1;
  else
    r = 0;
  return r;
}
```

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	✓

Transform the program, test suite, and template stmts into a program Q having:

1. the template code with the params $\boxed{c_i}$ represented as *global vars*
2. a location L guarded by conditional expressions representing test cases

Objective: finding values for $\boxed{c_i}$ allowing Q to reach L

Example: Solving Reachability Instance

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

int isUpP(int in,int up,int down){
  int bias, r;
  if (in)
    //template stmt
    bias = c0+c1bias+c2in+c3up+c4down;
  else
    bias = up;
  if (bias > down)
     r = 1;
  else
     r = 0;
  return r;
}
```

```
int main() {
   if(isUpP(1, 0,100) == 0 &&
      isUpP(1, 11,110) == 1 &&
      isUpP(0,100, 50) == 1 &&
      isUpP(1,-20, 60) == 1 &&
      isUpP(0, 0, 10) == 0 &&
      isUpP(0, 0,-10) == 1){

     [L] //target location
   }
   return 0;
}
```

Example: Solving Reachability Instance

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

int isUpP(int in,int up,int down){
  int bias, r;
  if (in)
    //template stmt
    bias = c0+c1bias+c2in+c3up+c4down;
  else
    bias = up;
  if (bias > down)
     r = 1;
  else
     r = 0;
  return r;
}
```

```
int main() {
   if(isUpP(1, 0,100) == 0 &&
      isUpP(1, 11,110) == 1 &&
      isUpP(0,100, 50) == 1 &&
      isUpP(1,-20, 60) == 1 &&
      isUpP(0, 0, 10) == 0 &&
      isUpP(0, 0,-10) == 1){

     [L] //target location
   }
   return 0;
}
```

Reaching L means fixing the original program

- L is reachable iff program passes all (6) tests
- Use values found for $\boxed{c_i}$ to represent new (repaired) stmts

Example: Solving Reachability Instance

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

int isUpP(int in,int up,int down){
  int bias, r;
  if (in)
    //template stmt
    bias = c0+c1bias+c2in+c3up+c4down;
  else
    bias = up;
  if (bias > down)
     r = 1;
  else
     r = 0;
  return r;
}
```

```
int main() {
   if(isUpP(1, 0,100) == 0 &&
      isUpP(1, 11,110) == 1 &&
      isUpP(0,100, 50) == 1 &&
      isUpP(1,-20, 60) == 1 &&
      isUpP(0, 0, 10) == 0 &&
      isUpP(0, 0,-10) == 1){

     [L] //target location
   }
   return 0;
}
```

Reaching L means fixing the original program

- L is reachable iff program passes all (6) tests
- Use values found for $\boxed{c_i}$ to represent new (repaired) stmts

Apply an off-the-self test-input generation tool

- E.g., KLEE determines $c_0 = 100, c_1 = 0, c_2 = 0, c_3 = 1, c_4 = 0$
- Map to new stmt `bias = 100 + up;` that fixes the orig program

Preliminaries

- Consider imperative language like C
- Include usual constructs (e.g., assignments, conditionals, loops, functions)
- Input a (potentially empty) tuple of values and output a value
- Consist of a finite set of functions, including a starting function main_p
- Semantics are specified by a test suite of finite input/output pairs
- For simplicity, assume support for exceptions (e.g., C++ and Java)

Reachability Problem

- Determines if a program can reach a given location
- Used in many verification tasks:
  - Model checking: check reachability of states representing bad behaviors
  - Test-input generation: produce inputs giving high program coverage
- Undecidable in general

Reachability Problem

- Determines if a program can reach a given location
- Used in many verification tasks:
  - Model checking: check reachability of states representing bad behaviors
  - Test-input generation: produce inputs giving high program coverage
- Undecidable in general

Definition:

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?

Reachability Problem

- Determines if a program can reach a given location
- Used in many verification tasks:
  - Model checking: check reachability of states representing bad behaviors
  - Test-input generation: produce inputs giving high program coverage
- Undecidable in general

Definition:

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?

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

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

  return 0;
}

//reaches L using x=-20,y=-40
```

Template-based Synthesis

- Practical synthesis techniques work on partially-complete programs and generate code from specific forms or templates
- *Template*: express shape of program constructs and contain holes (template params), e.g., $\boxed{c_0} + \boxed{c_1} v_1 + \boxed{c_2} v_2$
- *Template program* contains stmts with templates (e.g., $s := \boxed{c_0} + \ldots$)
- To synthesize template programs, existing techniques
  - Encode the program *and the specs* as a logical formula f
  - Use constraint solvers to find param values for c_i satisfying f
  - Instantiate templates with the param values to produce complete program

Template-based Synthesis

- Practical synthesis techniques work on partially-complete programs and generate code from specific forms or templates
- *Template*: express shape of program constructs and contain holes (template params), e.g., $\boxed{c_0} + \boxed{c_1} v_1 + \boxed{c_2} v_2$
- *Template program* contains stmts with templates (e.g., $s := \boxed{c_0} + \ldots$)
- To synthesize template programs, existing techniques
    - Encode the program *and the specs* as a logical formula f
    - Use constraint solvers to find param values for c_i satisfying f
    - Instantiate templates with the param values to produce complete program

Definition:

Given a template program Q with a finite set of template params $S = \{\boxed{c_1}, \ldots, \boxed{c_n}\}$ and a finite test suite of input/output pairs $T = \{(i_1, o_1), \ldots, (i_m, o_m)\}$, do there exist param values c_i such that $\forall (i, o) \in T \,.\, (Q[c_1, \ldots, c_n])(i) = o$?

Template-based Synthesis

- Practical synthesis techniques work on partially-complete programs and generate code from specific forms or templates
- *Template*: express shape of program constructs and contain holes (template params), e.g., $\boxed{c_0} + \boxed{c_1} v_1 + \boxed{c_2} v_2$
- *Template program* contains stmts with templates (e.g., $s := \boxed{c_0} + \ldots$)
- To synthesize template programs, existing techniques
  - Encode the program *and the specs* as a logical formula f
  - Use constraint solvers to find param values for c_i satisfying f
  - Instantiate templates with the param values to produce complete program

Definition:

Given a template program Q with a finite set of template params $S = \{\boxed{c_1}, \ldots, \boxed{c_n}\}$ and a finite test suite of input/output pairs $T = \{(i_1, o_1), \ldots, (i_m, o_m)\}$, do there exist param values c_i such that $\forall (i, o) \in T \, . \, (Q[c_1, \ldots, c_n])(i) = o$?

```
int isUp(int in,int up,int down){
  int bias, r;
  if (in)
    //template stmt                          // Test suite
    bias = c0+c1bias+c2in+c3up+c4down;  Q(1,0,100)=0
  else                                     Q(1,11,110)=1
    bias = up;                             Q(0,100,50)=1
  if (bias > down) r = 1;                  Q(1,-20,60)=1
  else r = 0;                              Q(0,0,10)=0
  return r;                                Q(0,0,-10)=1
}
/*Passes test suite using
c_0=100,c_1=1,c_2=0,c_3=1,c_4=0*/
```

Theorem 1: Template-based synthesis is Reduciable to Reachability

Reduction Sketch: *GadgetS2R* reduces a template-based synthesis instance $(Q, S = \{\boxed{c_i}, \ldots, \boxed{c_n}\}, T = \{(i_1, o_1), \ldots\})$ to a specific reachability problem (P, L), satisfiabled iff the synthesis instance can be solved

Theorem 1: Template-based synthesis is Reduciable to Reachability

Reduction Sketch: *GadgetS2R* reduces a template-based synthesis instance $(Q, S = \{\boxed{c_i}, \dots, \boxed{c_n}\}, T = \{(i_1, o_1), \dots\})$ to a specific reachability problem (P, L), satisfiabled iff the synthesis instance can be solved

- For every template param $\boxed{c_i}$, create a fresh global variables v_i

Theorem 1: Template-based synthesis is Reduciable to Reachability

Reduction Sketch: *GadgetS2R* reduces a template-based synthesis instance $(Q, S = \{\boxed{c_i}, \ldots, \boxed{c_n}\}, T = \{(i_1, o_1), \ldots\})$ to a specific reachability problem (P, L), satisfiabled iff the synthesis instance can be solved

- For every template param $\boxed{c_i}$, create a fresh global variables v_i
- For every fun $q \in Q$, define a similar fun $q_P \in P$ that has every reference to a c_i replaced with the corresponding var v_i

Theorem 1: Template-based synthesis is Reduciable to Reachability

Reduction Sketch: *GadgetS2R* reduces a template-based synthesis instance $(Q, S = \{\boxed{c_i}, \ldots, \boxed{c_n}\}, T = \{(i_1, o_1), \ldots\})$ to a specific reachability problem (P, L), satisfiabled iff the synthesis instance can be solved

- For every template param $\boxed{c_i}$, create a fresh global variables v_i
- For every fun $q \in Q$, define a similar fun $q_P \in P$ that has every reference to a c_i replaced with the corresponding var v_i
- The test suite T as a conjunctive expression e:

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

Theorem 1: Template-based synthesis is Reduciable to Reachability

Reduction Sketch: *GadgetS2R* reduces a template-based synthesis instance $(Q, S = \{\boxed{c_i}, \ldots, \boxed{c_n}\}, T = \{(i_1, o_1), \ldots\})$ to a specific reachability problem (P, L), satisfiabled iff the synthesis instance can be solved

- For every template param $\boxed{c_i}$, create a fresh global variables v_i
- For every fun $q \in Q$, define a similar fun $q_P \in P$ that has every reference to a c_i replaced with the corresponding var v_i
- The test suite T as a conjunctive expression e:

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

- Starting function $main_P$ consisting of conditional stmt leading to a target location L iff e is true

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

Theorem 1: Template-based synthesis is Reduciable to Reachability

Reduction Sketch: *GadgetS2R* reduces a template-based synthesis instance $(Q, S = \{\boxed{c_i}, \ldots, \boxed{c_n}\}, T = \{(i_1, o_1), \ldots\})$ to a specific reachability problem (P, L), satisfiabled iff the synthesis instance can be solved

- For every template param $\boxed{c_i}$, create a fresh global variables v_i
- For every fun $q \in Q$, define a similar fun $q_P \in P$ that has every reference to a c_i replaced with the corresponding var v_i
- The test suite T as a conjunctive expression e:

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

- Starting function $main_P$ consisting of conditional stmt leading to a target location L iff e is true

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

- P consists of the new variables v_i, funs q_p, and $main_p$

Example:

```
int isUp(int in,int up,int down){
  int bias, r;
  if (in)
    bias = c0+c1bias+c2in+c3up+c4down;
  else
    bias = up;
  if (bias > down) r = 1;
  else r = 0;
  return r;
}
// Test suite
Q(1,0,100)=0;  Q(1,-20,60)=1
Q(1,11,110)=1; Q(0,0,10)=0
Q(0,100,50)=1; Q(0,0,-10)=1
```

$\longrightarrow$

```
int c0,c1,c2,c3,c4; //global inputs
int isUpP(int in,int up,int down){
  int bias, r;
  if (in)
    bias = c0+c1bias+c2in+c3up+c4down;
  else
    bias = up;
  if (bias > down) r = 1;
  else r = 0;
  return r;
}
```

```
int main() {
  if(isUpP(1, 0,100) == 0 &&
     isUpP(1, 11,110) == 1 &&
     isUpP(0,100, 50) == 1 &&
     isUpP(1,-20, 60) == 1 &&
     isUpP(0, 0, 10) == 0 &&
     isUpP(0, 0,-10) == 1){

    [L] //target location
  }
  return 0;
}
```

Input: Template-based instance

Output: Reachability instance

Example:

```
int isUp(int in,int up,int down){
  int bias, r;
  if (in)
    bias = c0+c1bias+c2in+c3up+c4down;
  else
    bias = up;
  if (bias > down) r = 1;
  else r = 0;
  return r;
}
// Test suite
Q(1,0,100)=0;  Q(1,-20,60)=1
Q(1,11,110)=1; Q(0,0,10)=0
Q(0,100,50)=1; Q(0,0,-10)=1
```

→

```
int c0,c1,c2,c3,c4; //global inputs
int isUpP(int in,int up,int down){
  int bias, r;
  if (in)
    bias = c0+c1bias+c2in+c3up+c4down;
  else
    bias = up;
  if (bias > down) r = 1;
  else r = 0;
  return r;
}
```

```
int main() {
  if(isUpP(1, 0,100) == 0 &&
     isUpP(1, 11,110) == 1 &&
     isUpP(0,100, 50) == 1 &&
     isUpP(1,-20, 60) == 1 &&
     isUpP(0, 0, 10) == 0 &&
     isUpP(0, 0,-10) == 1){

    [L] //target location
  }
  return 0;
}
```

Input: Template-based instance

Output: Reachability instance

$GadgetS2R : (Q, S, T) \mapsto (P, L)$

- Correctness: relies on two key invariants
  1. funs $\in P$ have the same behavior as funs $\in Q$
  2. L is reachable iff values of c_i can be assigned to v_i allowing Q to pass all tests
- Complexity: *linear* in program size and test cases of synthesis instance

Theorem 2: Reachability is reduciable to Template-based synthesis

Reduction Sketch: *GadgetR2S* reduces a reachability problem (P, L) to a specific template-based synthesis instance $(Q, S = \{\boxed{c_i}, \ldots, \boxed{c_n}\}, T = \{(i_1, o_1), \ldots\})$, solvable iff the reachability problem can be satisfied

Theorem 2: Reachability is reduciable to Template-based synthesis

Reduction Sketch: *GadgetR2S* reduces a reachability problem (P, L) to a specific template-based synthesis instance $(Q, S = \{\boxed{c_i}, \ldots, \boxed{c_n}\}, T = \{(i_1, o_1), \ldots\})$, solvable iff the reachability problem can be satisfied

- For every variable v_i, define a fresh template variables $\boxed{c_i}$. The set S of template params contain each $\boxed{c_i}$

Theorem 2: Reachability is reduciable to Template-based synthesis

Reduction Sketch: *GadgetR2S* reduces a reachability problem (P, L) to a specific template-based synthesis instance $(Q, S = \{\boxed{c_i}, \ldots, \boxed{c_n}\}, T = \{(i_1, o_1), \ldots\})$, solvable iff the reachability problem can be satisfied

- For every variable v_i, define a fresh template variables $\boxed{c_i}$. The set S of template params contain each $\boxed{c_i}$
- For every fun $p \in P$, define a similar fun $p_Q \in Q$
  - Replace each call to p with a corresponding call to p_Q
  - Replace each use of v_i with a read from c_i

Theorem 2: Reachability is reduciable to Template-based synthesis

Reduction Sketch: *GadgetR2S* reduces a reachability problem (P, L) to a specific template-based synthesis instance $(Q, S = \{\boxed{c_i}, \ldots, \boxed{c_n}\}, T = \{(i_1, o_1), \ldots\})$, solvable iff the reachability problem can be satisfied

- For every variable v_i, define a fresh template variables $\boxed{c_i}$. The set S of template params contain each $\boxed{c_i}$
- For every fun $p \in P$, define a similar fun $p_Q \in Q$
  - Replace each call to p with a corresponding call to p_Q
  - Replace each use of v_i with a read from c_i
- Raise a unique *exception* `REACHED` at the loc $\in Q$ corresponding to $L \in P$

Theorem 2: Reachability is reduciable to Template-based synthesis

Reduction Sketch: *GadgetR2S* reduces a reachability problem (P, L) to a specific template-based synthesis instance $(Q, S = \{\boxed{c_i}, \ldots, \boxed{c_n}\}, T = \{(i_1, o_1), \ldots\})$, solvable iff the reachability problem can be satisfied

- For every variable v_i, define a fresh template variables $\boxed{c_i}$. The set S of template params contain each $\boxed{c_i}$
- For every fun $p \in P$, define a similar fun $p_Q \in Q$
  - Replace each call to p with a corresponding call to p_Q
  - Replace each use of v_i with a read from c_i
- Raise a unique *exception* `REACHED` at the loc $\in Q$ corresponding to $L \in P$
- Starting function $main_Q$ that returns 1 (indicating when `REACHED` is caught)

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

Theorem 2: Reachability is reduciable to Template-based synthesis

Reduction Sketch: *GadgetR2S* reduces a reachability problem (P, L) to a specific template-based synthesis instance $(Q, S = \{\boxed{c_i}, \ldots, \boxed{c_n}\}, T = \{(i_1, o_1), \ldots\})$, solvable iff the reachability problem can be satisfied

- For every variable v_i, define a fresh template variables $\boxed{c_i}$. The set S of template params contain each $\boxed{c_i}$
- For every fun $p \in P$, define a similar fun $p_Q \in Q$
  - Replace each call to p with a corresponding call to p_Q
  - Replace each use of v_i with a read from c_i
- Raise a unique *exception* `REACHED` at the loc $\in Q$ corresponding to $L \in P$
- Starting function $main_Q$ that returns 1 (indicating when `REACHED` is caught)

```
int mainQ() {
  try {mainPQ();}
  catch (REACHED) {return 1;}
  return 0;
}
```

- Q consists of template parameters $S = \{\boxed{c_1}), \ldots, \boxed{c_n}\}$, funs p_Q's, and main_Q. The test suite T for Q consists of exactly one test case $Q() = 1$.

Example:

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

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

  return 0;
}
```

→

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

  return 0;
}
```

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

  return 0;
}
//Test suite: Q() = 1
```

Input: Reachability instance

Output: Template-based instance

Example:

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

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

  return 0;
}
```

→

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

  return 0;
}
```

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

  return 0;
}
//Test suite: Q() = 1
```

Input: Reachability instance

Output: Template-based instance

$GadgetR2S : (P, L) \mapsto (Q, S, T)$

- Correctness of *GadgetR2S* relies on two key invariants:
  1. For any c_i, execution in Q mirrors execution in P with $v_i \mapsto c_i$
  2. Q raises the exception REACHED iff P can reach L
- Complexity of *GadgetR2S* is *linear* in the input P, L (and $v_i \in P$)

Program Repair as a Synthesis Problem

Definition:

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 stmts $\{s_i\} \subseteq Q$ and parameter values $c_1, \ldots, c_n$ 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?

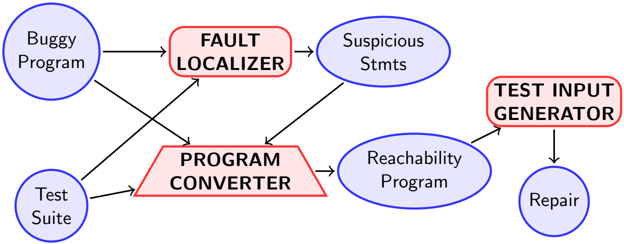
Program Repair as a Synthesis Problem

Definition:

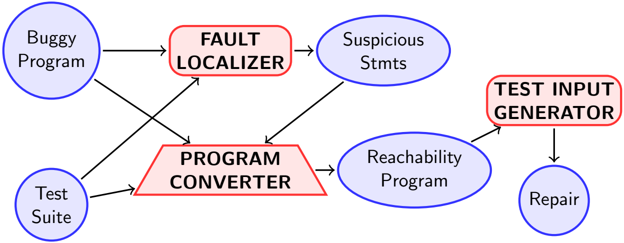
Given a program Q that *fails* at least one test in a finite test suite T and a finite set of parameterized templates S, does there exist a set of stmts $\{s_i\} \subseteq Q$ and parameter values $c_1, \ldots, c_n$ for the templates in S such that s_i can be replaced with $S[c_1, \ldots, c_n]$ and the resulting program passes all tests in T?

- Allows edits to multiple program stmts (e.g., can replace multilines with parameterized templates).
- *Single-edit* repair problem restricts the edits to one stmt

CETI: program repair using test-input generation



CETI: program repair using test-input generation



CETI: correcting errors using test-inputs

1. Use fault localization to obtain suspicious stmts
2. Apply synthesis templates to create template-based synthesis instances
3. Use reduction theorem to convert to reachability programs
4. Employ an off-the-shelf test-input generator to solve reachability, i.e., creating repairs

Repair Components

Repair Components

1. Fault Localization
   - Identify suspicious statements to modify (e.g., `bias = down;`)
   - Implement the statistical algorithm Tarantula to compute suspicious scores of program stmts (based on exe freq in passing and failing runs)

Repair Components

1. Fault Localization
   - Identify suspicious statements to modify (e.g., `bias = down;`)
   - Implement the statistical algorithm Tarantula to compute suspicious scores of program stmts (based on exe freq in passing and failing runs)
2. Synthesis Templates
   - Repair rhs of *assignment* stmts, e.g., $s := e \mapsto s :=$ parameterized template
   - Supports templates to modify constants, linear expressions, and logical, comparisons, and arithmetic ops, e.g.,

$$\begin{array}{lcl} s := x \leq y & \mapsto & s := x < y \\ s := x + y & \mapsto & s := x - y \\ s := 10 & \mapsto & s := 3x + 4y - 200 \end{array}$$

Repair Components

1. Fault Localization
   - Identify suspicious statements to modify (e.g., `bias = down;`)
   - Implement the statistical algorithm Tarantula to compute suspicious scores of program stmts (based on exe freq in passing and failing runs)
2. Synthesis Templates
   - Repair rhs of *assignment* stmts, e.g., $s := e \mapsto s :=$ parameterized template
   - Supports templates to modify constants, linear expressions, and logical, comparisons, and arithmetic ops, e.g.,

$$\begin{array}{lcl} s := x \leq y & \mapsto & s := x < y \\ s := x + y & \mapsto & s := x - y \\ s := 10 & \mapsto & s := 3x + 4y - 200 \end{array}$$

3. Test-input Generation
   - Employs the symbolic execution tool KLEE to find test inputs for C programs
   - Can easily be extended to use other verification tools such as CPAChecker, JPF, PEX, etc
   - Advantage of reduction: admit other reachability/verification tools, regardless of technologies used

Implementation and Evaluation

CETI

- Written in OCaml, employ CIL to parse and modify C programs
- Take as input a testsuite T, and a C program P that fails T, return modified statements allowing P to pass T
- Synthesize correct-by-construction repairs (guarantee to pass test suite)
- Currently support single-edit repairs (can be extended to multiline repairs by considering k susp stmts)

Implementation and Evaluation

CETI

- Written in OCaml, employ CIL to parse and modify C programs
- Take as input a testsuite T, and a C program P that fails T, return modified statements allowing P to pass T
- Synthesize correct-by-construction repairs (guarantee to pass test suite)
- Currently support single-edit repairs (can be extended to multiline repairs by considering k susp stmts)

Evaluation

- Evaluate using the TCAS program from SIR benchmark
  - Implement aircraft traffic collision avoidance system, 180 LoC's
  - About 1608 tests and 41 faulty (seeded defects) functions (changed operators, incorrect constant values, missing code, and incorrect control flow)
  - Have the most introduced defects (41) in SIR, and been used to benchmark modern bug repair techniques
- Use all available tests to guarantee that any repair found is correct wrt entire test suite

Experimental Results

- Fixed 26 of 41 defects, including multiple defects of different types, avg 22s
- Found 100% of repairs with single-edit changes under considered templates
- Found several ways to fix defects and also obtained unexpected repairs

Experimental Results

- Fixed 26 of 41 defects, including multiple defects of different types, avg 22s
- Found 100% of repairs with single-edit changes under considered templates
- Found several ways to fix defects and also obtained unexpected repairs
- Not able to repair 15 of 41 defects (require multiloc edits or code not under considered templates)

Experimental Results

- Fixed 26 of 41 defects, including multiple defects of different types, avg 22s
- Found 100% of repairs with single-edit changes under considered templates
- Found several ways to fix defects and also obtained unexpected repairs
- Not able to repair 15 of 41 defects (require multiloc edits or code not under considered templates)
- Performs well compared to other repair tools
    - GenProg, which uses a genetic algorithm, can repair 11 of these defects
    - FoREnSiC, which uses the concolic execution in CREST, repairs 23 defects
    - Other repair techniques, including random mutation, equivalence checking, and counterexample guided refinement, repair 9, 15 and 16 defects, respectively.
    - SemFix outperforms CETI, repairing 34 defects
- Existing approaches *integrates* verification techniques, CETI eschews heavyweight analyses
    - Reduction and offload synthesis to KLEE
    - Unoptimized CETI already performed reasonably well

Conclusions

Formal connection between template-based program synthesis and the reachability problem in verification

- Constructively prove an equivalence between these two problems
- Reduce template-based synthesis reachability, and conversely
- Connect the two problems and enable the application of ideas, optimizations, and tools developed for one problem to the other

CETI: new automatic program repair technique

- An algorithm and tool for repairing C programs using test-input generation
  - Transform the task of synthesizing program repairs to a reachability problem
  - Use off-the-shelf test-input generation tool to synthesize repaired stmts
  - Achieve higher success rates than many other standard repair approaches.

`https://bitbucket.org/nguyenthanhvuh/ceti`