Symbolic Execution for Automated Repair

[Prof. Abhik Roychoudhury]

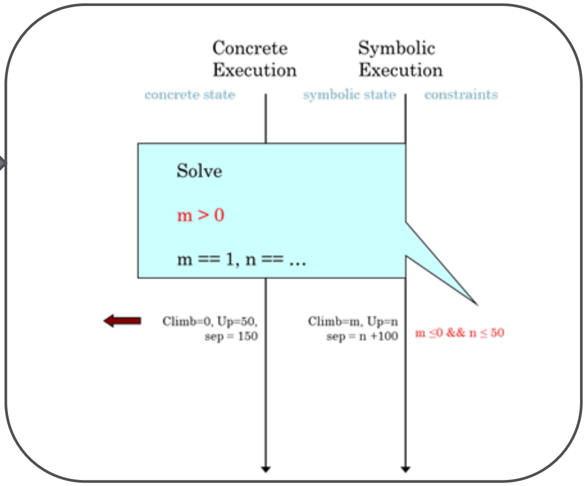
National University of Singapore

abhik@comp.nus.edu.sg

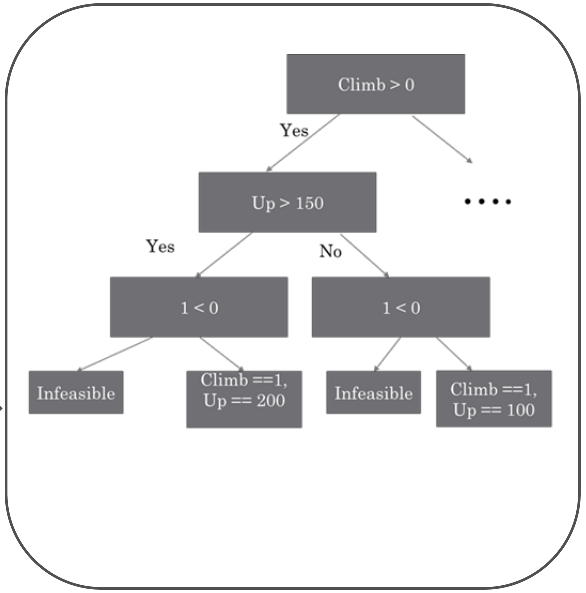
Recap: Use of Symbolic Execution

Bug Finding

- Concolic execution: supporting *real* executions [Directed Automated Random Testing]

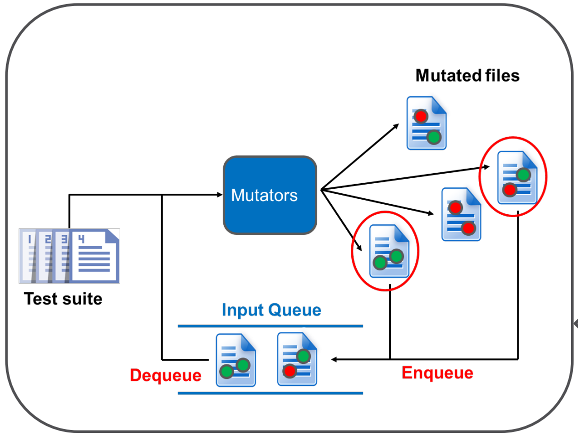


- Symbolic execution tree construction e.g. KLEE [Modeling system environment]



- Grey-box fuzz testing for systematic path exploration inspired by concolic execution

AFLFast

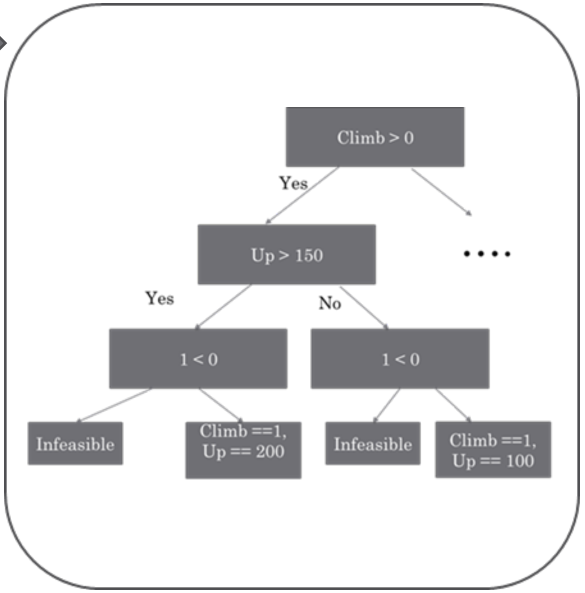


Recap: Use of Symbolic Execution

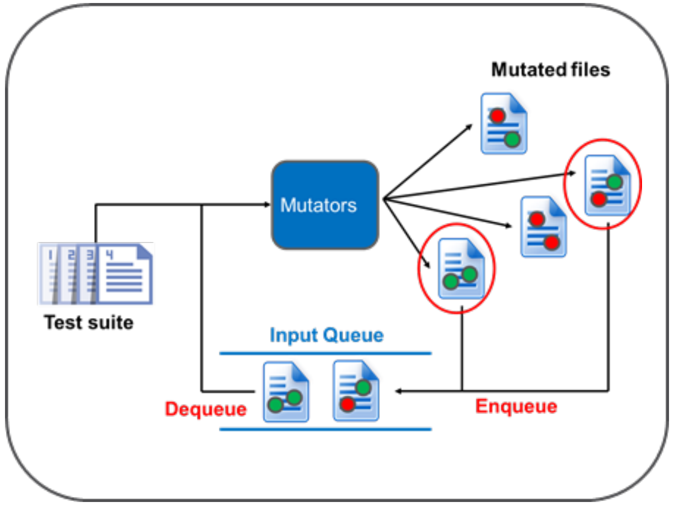
Reachability Analysis

Reachability of a location in the program

- Traverse the symbolic execution tree using search strategies e.g. KATCH



- Encode it as an optimization problem inside the genetic search of grey-box fuzzing AFLGo

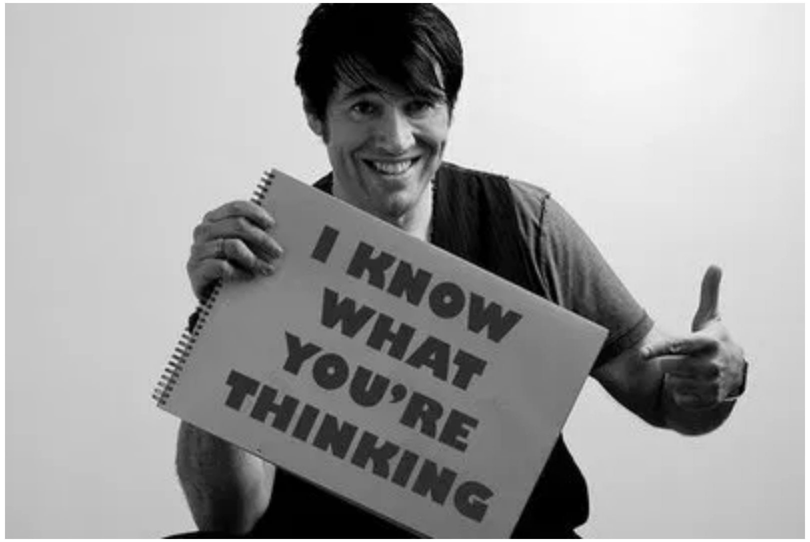


Reflections on Symbolic Execution

Specification Inference (TODAY!)

(application: localization, repair)

In the absence of formal specifications, analyze the buggy program and its artifacts such as execution traces via various heuristics to glean a specification about how it can pass tests and what could have gone wrong!



Bug Fixing

- Most software has many bugs.
- Security-related bugs should be fixed before they are exploited by malicious users.
- Oftentimes, bugs are not fixed even a few months after they were reported.
- E.g. Bug 18665 of glibc
  - Reported and responded on July 2015
  - Patched on Feb 2016
  - CVSS score: 8.1 / 10 (buffer overflow)
- *"Thanks for the bug report. Do you have a **test case** that triggers this scenario? Do you have a **patch** or suggested fix?"*

Background

- Why debugging is hard?
  - Huge search space ? OR …

- What would make debugging easy?
  - Specification Inference

- Ideas in debugging which lead **to automated fixing**
  - Using implicit specification inference.

A quote from many years ago

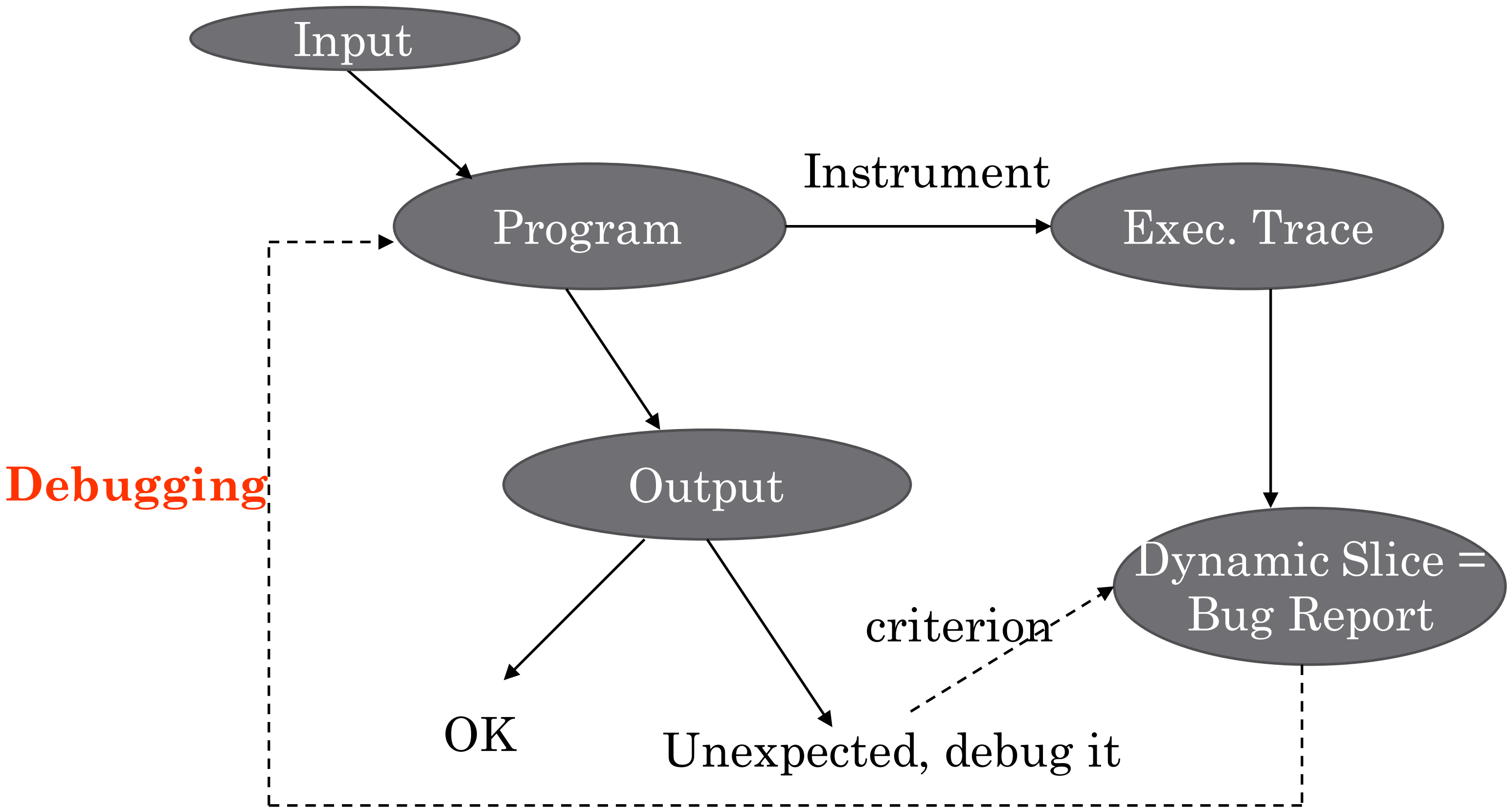
"Even today, debugging remains very much of an art. Much of the computer science community has largely ignored the debugging problem..... over 50 percent of the problems resulted from the time and space chasm between symptom and root cause or inadequate debugging tools."

Hailpern & Santhanam, IBM Systems Journal, 41(1), 2002

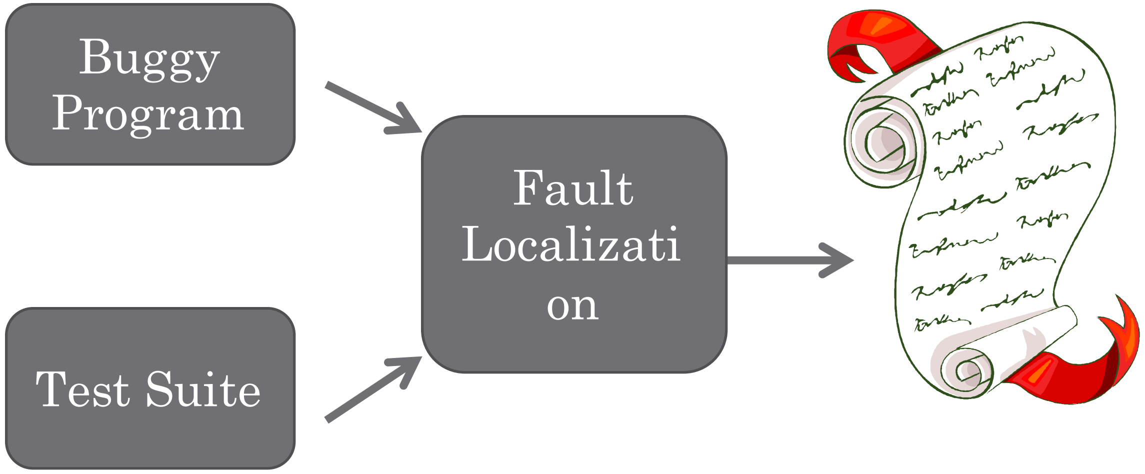
Any progress in 2002 – 2018?

How can symbolic execution help?

Dynamic slicing: a debugging aid



Statistical Fault localization



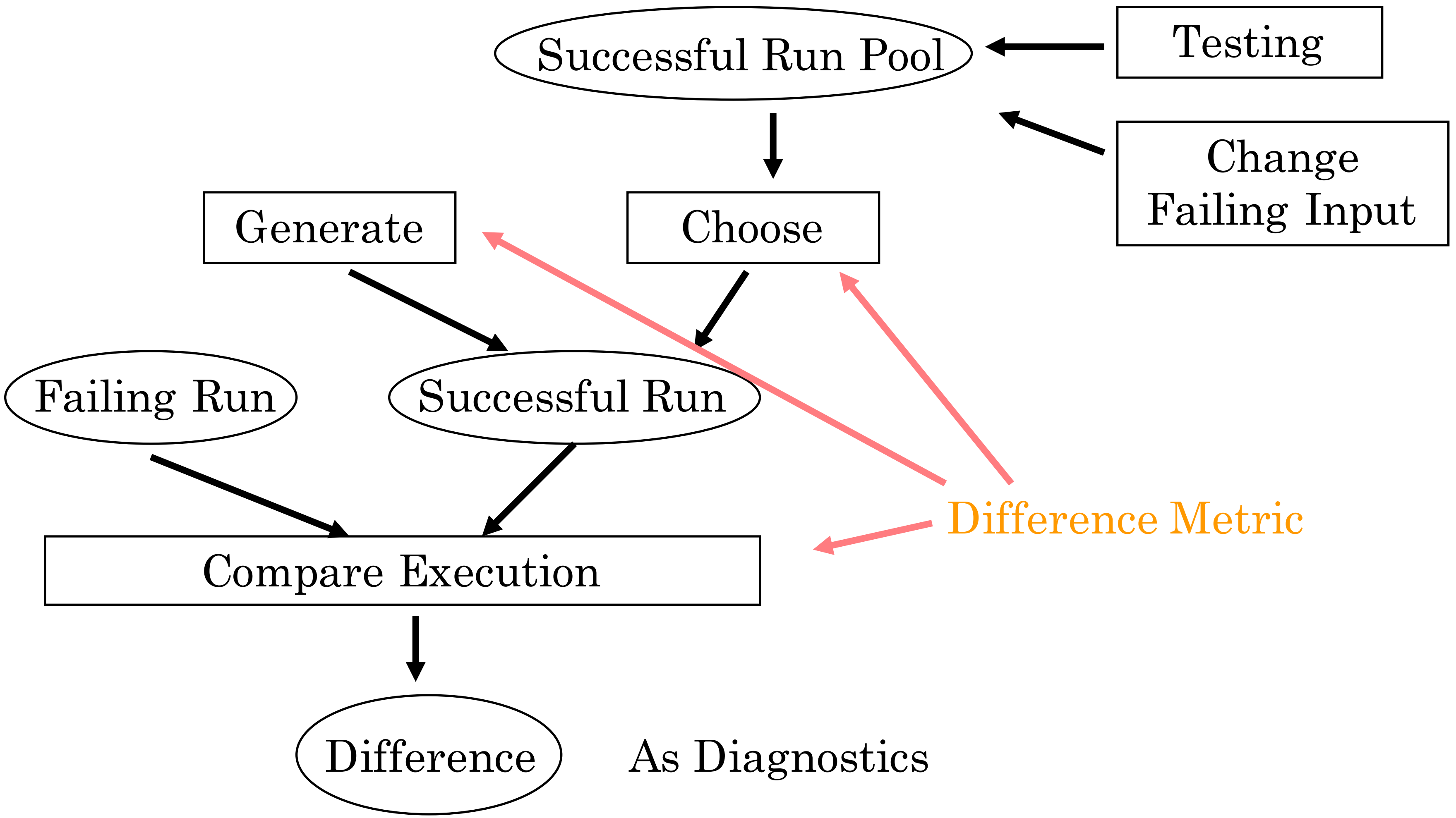
Ranked list of suspicious statements

Assign scores to program statements based on their occurrence in passing / failing tests. ***Correlation equals causation!***

$$\text{Score(s)} = \frac{\frac{\text{fail(s)}}{\text{allfail}}}{\frac{\text{fail(s)}}{\text{allfail}} + \frac{\text{pass(s)}}{\text{allpass}}}$$

An example of scoring scheme [Tarantula]

Trace Comparison based Debugging



A moment's note for the students

- You have – buggy program, failing tests
- You do not have specification of intended behavior, try to discover
- [What the program is supposed to do]

- Compare this to software model checking
  - You have formal specification of intended behavior (temporal logic property)
  - You have the buggy program
  - You do not have failing tests (counter-examples), try to discover.

What is the intended behavior?

Only in the programmer's mind?
Assertions capturing programmer's intent at each statement
Too much overhead on programmer: almost as much work as a proof

Source of Information	Name of Symbolic Technique
Internal inconsistency	Cause Clue Clauses [PLDI 11] Error Invariants [FM 12]
Passing Tests	Angelic Debugging [ICSE 11]
Previous version / Golden implementation	Regression Debugging [FSE09, FSE10, FSE11]

Example

```
Input: a, index
base = a;
sentinel = base;
offset = index;
address = base + offset;
output address, sentinel
```

Test 1
<a, index==10>
assert sentinel <= address
assert address < a + 10

Test 2
<a, index==9>
assert sentinel <= address
assert address < a + 10

CCC : General idea

<a, index == 10> $\wedge$

```
Input: a, index
1.   base = a;
2.   sentinel = base;
3.   offset = index;
4.   address = base +
     offset;
5.   assert sentinel <=
     address
```

$\wedge$ address $< a + 10$ == false

Failing input $\wedge$ Program formula $\wedge$ Observed Output == false

Cause Clue Clauses, Jose and Majumdar, PLDI 2011.

CCC: General idea

<a, index == 10> $\wedge$

```
Input: a, index
1.   base = a;
2.   sentinel = base;
3.   offset = index;
4.   address = base +
     offset;
5.   assert sentinel <=
     address
```

$\wedge$ address < a + 10 == false

index== 10 $\wedge$ base == a $\wedge$ sentinel == base $\wedge$ offset == index $\wedge$ address == base + offset $\wedge$ sentinel $\leq$ address $\wedge$ **address < a + 10** == false

Hard constraint

Soft constraint

Hard constraint

First iteration

Hard constraint		Soft constraint		Hard constraint		
index== 10	$\wedge$	base == a $\wedge$ sentinel == base $\wedge$ offset == index $\wedge$ address == base + offset $\wedge$ sentinel $\leq$ address	$\wedge$	**address < a + 10**	==	false

Running Partial MAXSAT, we get base == a as a soft constraint that can be removed.

Corresponds to the fix:

```
Input: a, index
base = a - 1;
sentinel = base;
offset = index;
address = base + offset;
output address, sentinel
```

Moving further

Hard constraint

Hard & Soft constraints

Hard constraint

index= 10 $\wedge$ ***base == a*** $\wedge$ sentinel == base $\wedge$ offset == index $\wedge$ address == base + offset $\wedge$ sentinel $\leq$ address $\wedge$ **address < a + 10** == false

We mark ***base == a*** as hard now, and run Partial MaxSAT again, to get offset == index.

Corresponds to the fix:

```
Input: a, index
1.   base = a;
2.   sentinel = base;
3.   offset = index - 1;
4.   address = base + offset;
5.   output address, sentinel
```

The clause sentinel==base does not help (or hurt)

Fix determines fault

```
Input: a, index
1.   base = a;
2.   sentinel = base;
3.   offset = index;
4.   address = base + offset;
5.   output address, sentinel
```

Off-by-one error

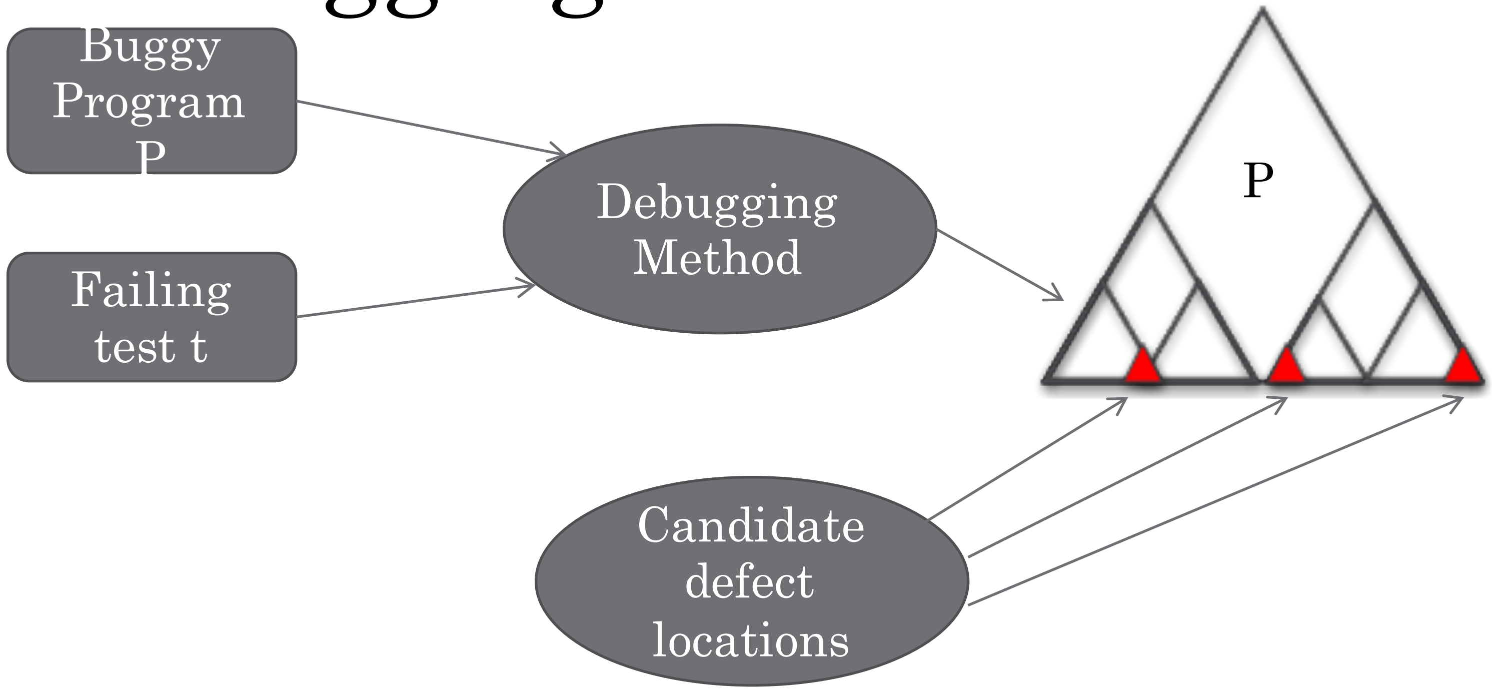
```
Input: a, index
1.   base = a - 1;
2.   sentinel = base;
3.   offset = index;
4.   address = base + offset;
5.   output address, sentinel
```

```
Input: a, index
1.   base = a;
2.   sentinel = base;
3.   offset = index - 1;
4.   address = base + offset;
5.   output address, sentinel
```

Specification discovery?

- Find statements that cause inconsistency in the failing execution
  - Removal of that inconsistency makes the error go away
    - Minimal inconsistency → cause
  - Starting point for repair
  - Simple specification discovery
    - Removing statement S causes error to disappear
    - Do not know what S should have been!

Angelic Debugging



Define possible defect locations by identifying expressions which can fix the fault!

Angelic Debugging, ICSE 2011

Ack: Satish Chandra (Facebook)

General idea – fix failing tests,…

<a, index == 10> ∧

```
Input: a, index
base = a;
sentinel = base;
offset = index;
address = base + offset;
assert sentinel <= address
```

∧ assert address < a + 10 == false

index== 10 ∧ base == ?? ∧ sentinel == base ∧ offset == index ∧ address == base + offset ∧ sentinel <= address ∧ **address < a + 10**

base = a is a valid fix location.

Note: Does not suggest the repaired statement base = a – 1.

Fix failing tests, …

<a, index == 10> $\wedge$

```
Input: a, index
1.   base = a;
2.   sentinel = base;
3.   offset = index;
4.   address = base +
     offset;
5.   assert sentinel <=
     address
```

$\wedge$ assert address < a + 10 == false

index== 10 $\wedge$ base == a $\wedge$ sentinel == base $\wedge$ **offset == ??** $\wedge$ address == base + offset $\wedge$ sentinel <= address $\wedge$ **address < a + 10**

offset = index is another valid fix location.

…, and do not break passing tests

<a, index == 1> $\wedge$

```
Input: a, index
1.   base = a;
2.   sentinel = base;
3.   offset = index;
4.   address = base +
     offset;
5. output address,
sentinel
```

$\wedge$ assert sentinel == a

NEW

```
Input: a, index
1.   base = ??;
2.   sentinel = base;
3.   offset = index;
4.   address = base +
     offset;
5.   output address,
     sentinel
```

```
Input: a, index
1.   base = a;
2.   sentinel = base;
3.   offset = ??;
4.   address = base +
     offset;
5.   output address,
     sentinel
```

OK !!

Specification discovery?

- Passing tests tell us which expressions are “inflexible”
  - The better your test suite is, the more you know!
- Therefore, the bug must be in one of the flexible expressions
- *Limitations*
  - Assumption of 1-fixable
  - Quality of filtering depends on the goodness of test suite
  - Subject to implementation of the symbolic analysis

Retrospective

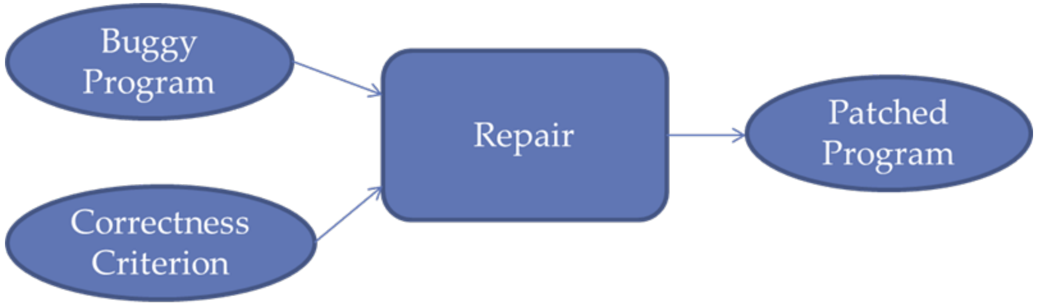
Debugging – some milestones

- Manual era: prints and breakpoints
- Statistical fault localization [e.g. Tarantula]
- Dynamic slicing [e.g. JSlice]
- Trace comparison and delta debugging
  - Look for workarounds – *how to avoid the error*?
- **Symbolic techniques**
  - Replace repeated experimentation with constraint solving.
  - Discover and (partially) infer intended semantics by symbolic analysis

- **The Future: repair (hints)**

Syntactic Program Repair

Automated Program Repair



- **[OLD]** *Large search space* of candidate patches for general-purpose repair tools.
- **[NEW]** Weak description of intended behavior / *correctness criterion* e.g. tests
- **[FUTURE]** Patch suggestions and *Interactive Repair*

Research Issues in Program Repair

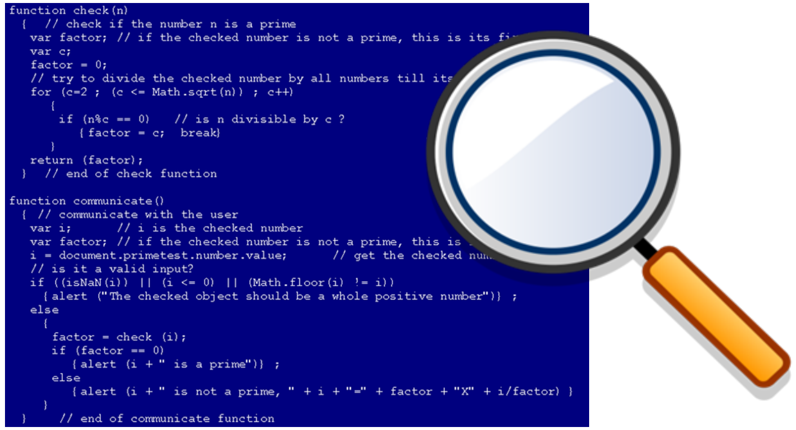
- [**OLD**] *Large search space* of candidate patches for general-purpose repair tools.
- ->. ***What should I use?***
- -> ***Which search frameworks could we use?***
- -> ***Syntactic Program Repair***

- [**NEW**] Weak description of intended behavior / *correctness criterion* e.g. tests
- -> ***Overfitting of a patch candidate to tests?***
- -> ***Extract specification from test executions to reduce overfitting.***
- ->. ***Do so, while still navigating the search space***
- -> ***Semantic Program Repair***

Division of Labor

Syntactic Program Repair

Semantic Program Repair



Syntactic Program Repair:

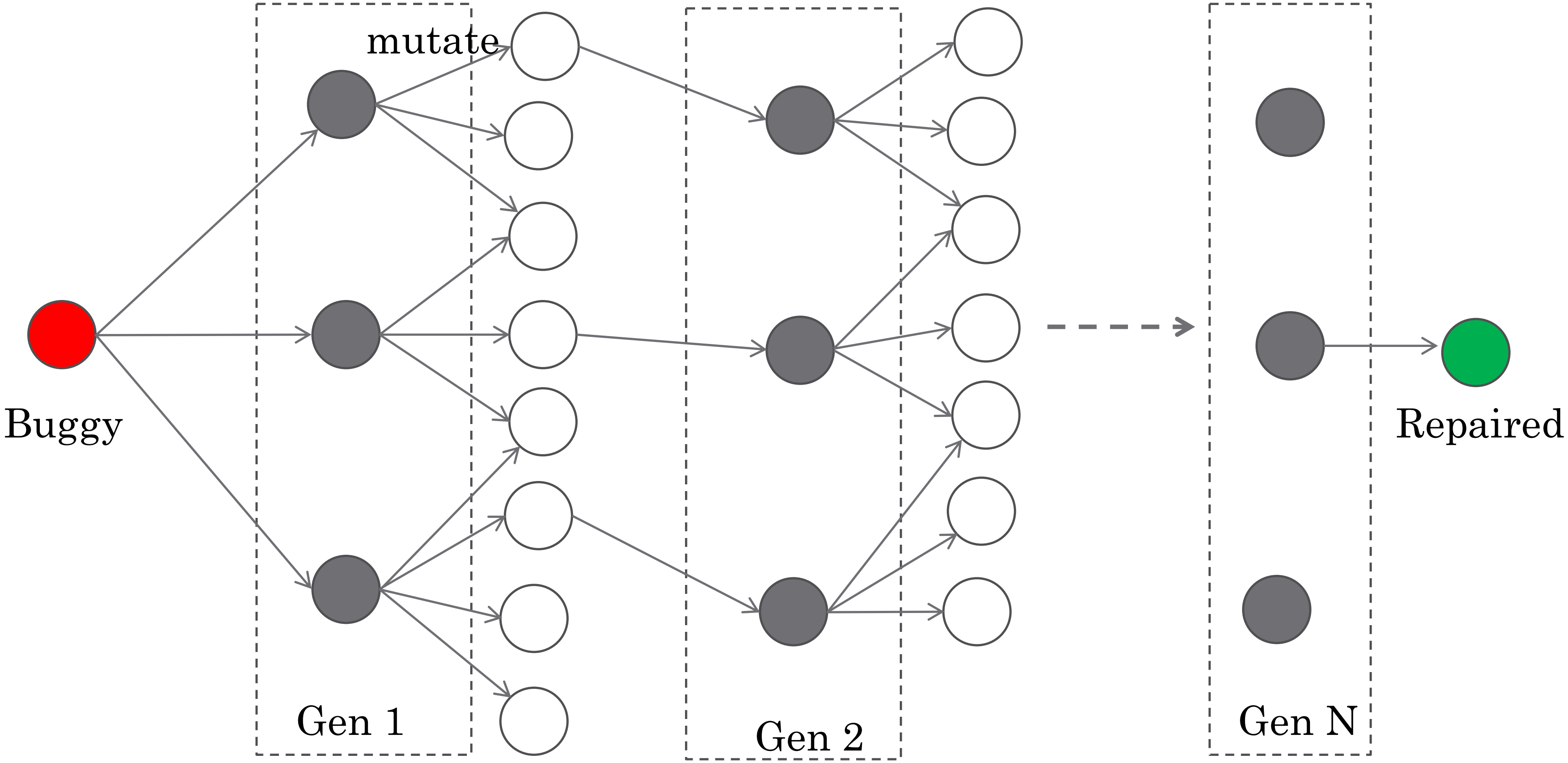
1. Where to fix, which line?
2. Generate patches in the candidate line
3. Validate the candidate patches against correctness criterion.

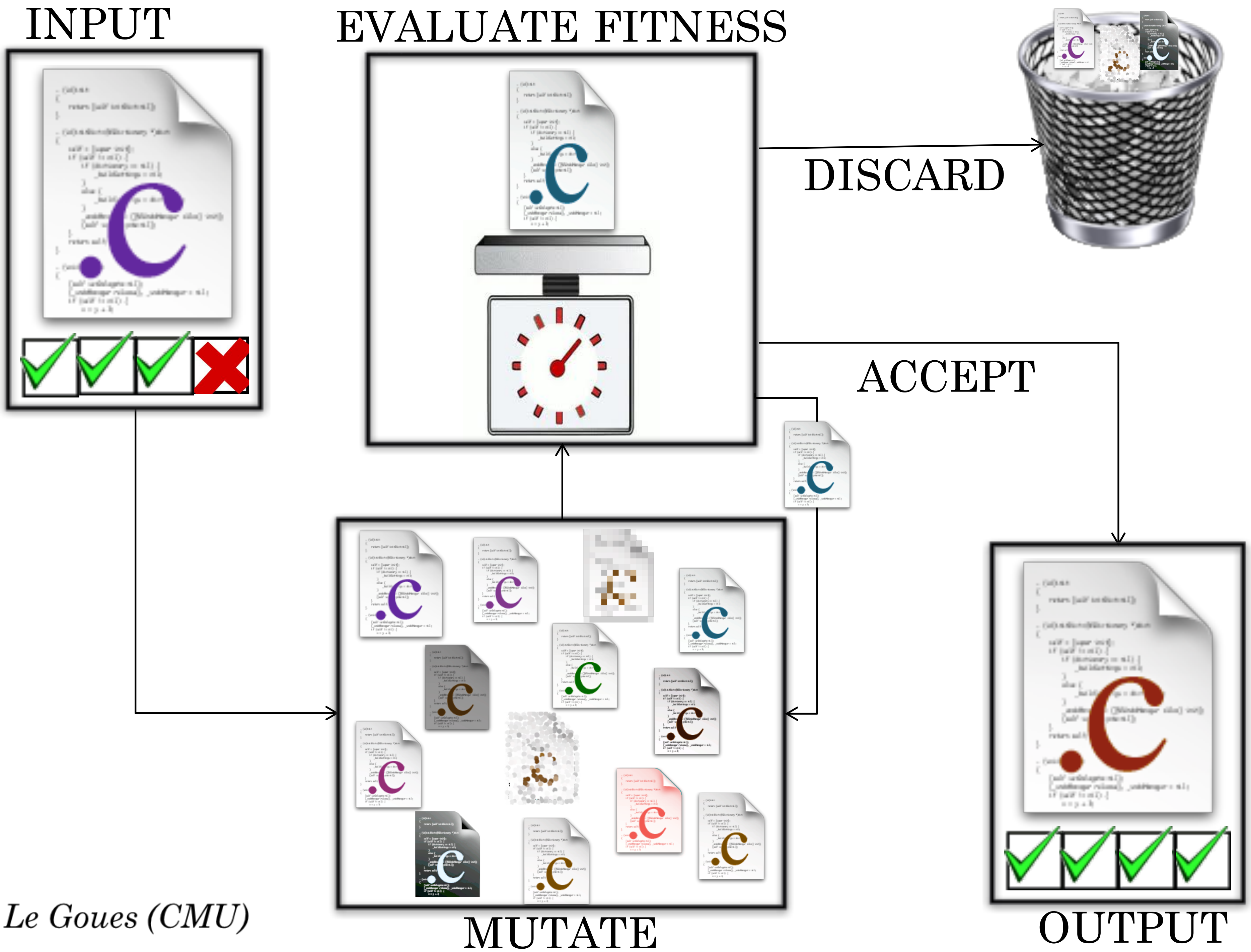
Semantic Program Repair:

1. Where to fix, which line(s)?
2. What values should be returned by those lines, e.g. <inp ==1, ret== 0>
3. What are the expressions which will return such values?

ISSISP Summer School 2018

GenProg – repair via search *(Ack: Claire Le Goues)*





Ack: Claire Le Goues (CMU)

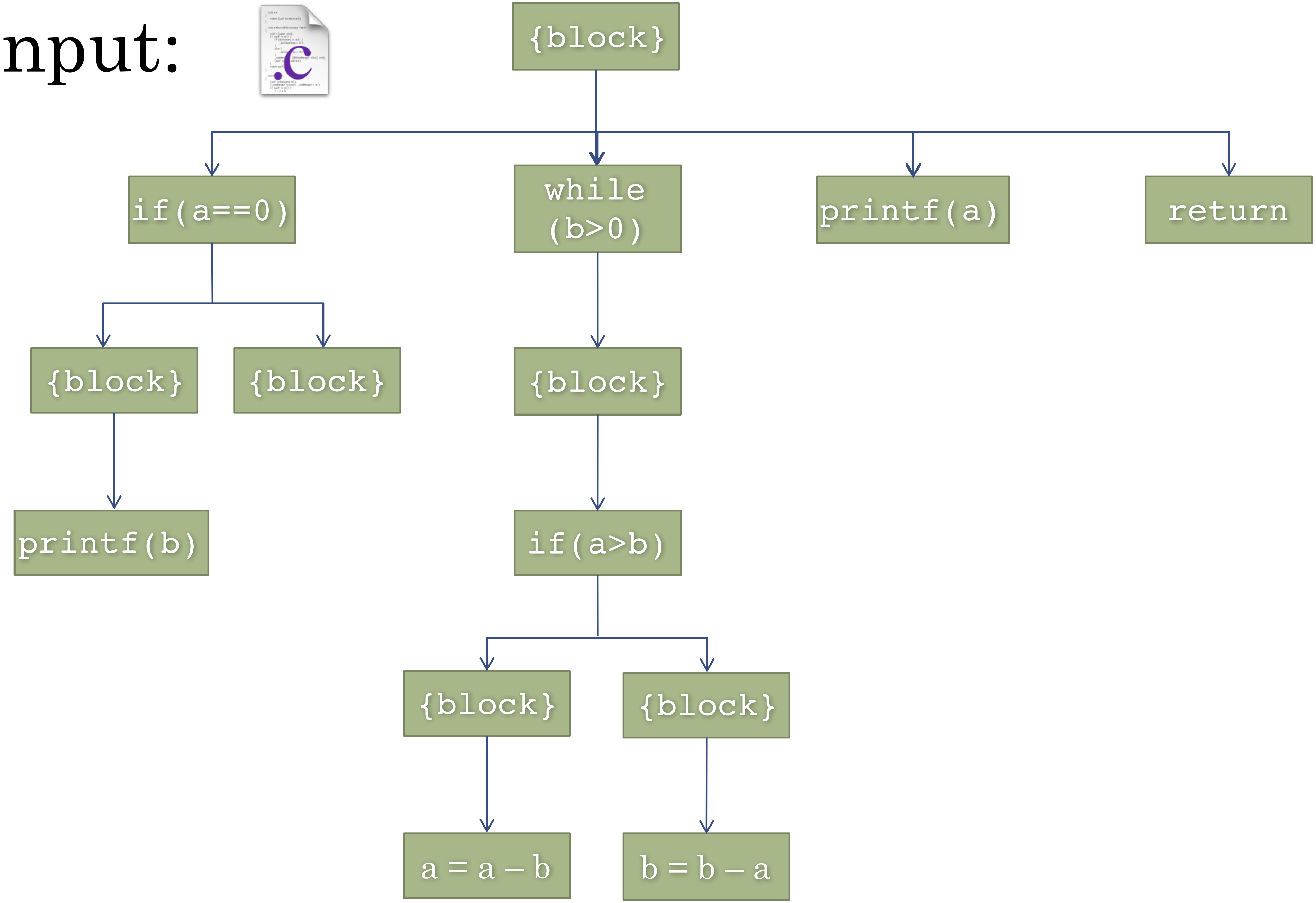
```
> gcd(4,2)
> 2
>
> gcd(1071,1029)
> 21
>
> gcd(0,55)
> 55
```

(looping forever)

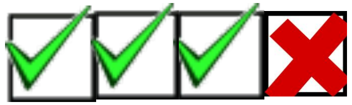
```c
void gcd(int a, int b) {
  if (a == 0) {
    printf("%d", b);
  }
  while (b > 0) {
    if (a > b)
      a = a – b;
    else
      b = b – a;
  }
  printf("%d", a);
  return;
}
```

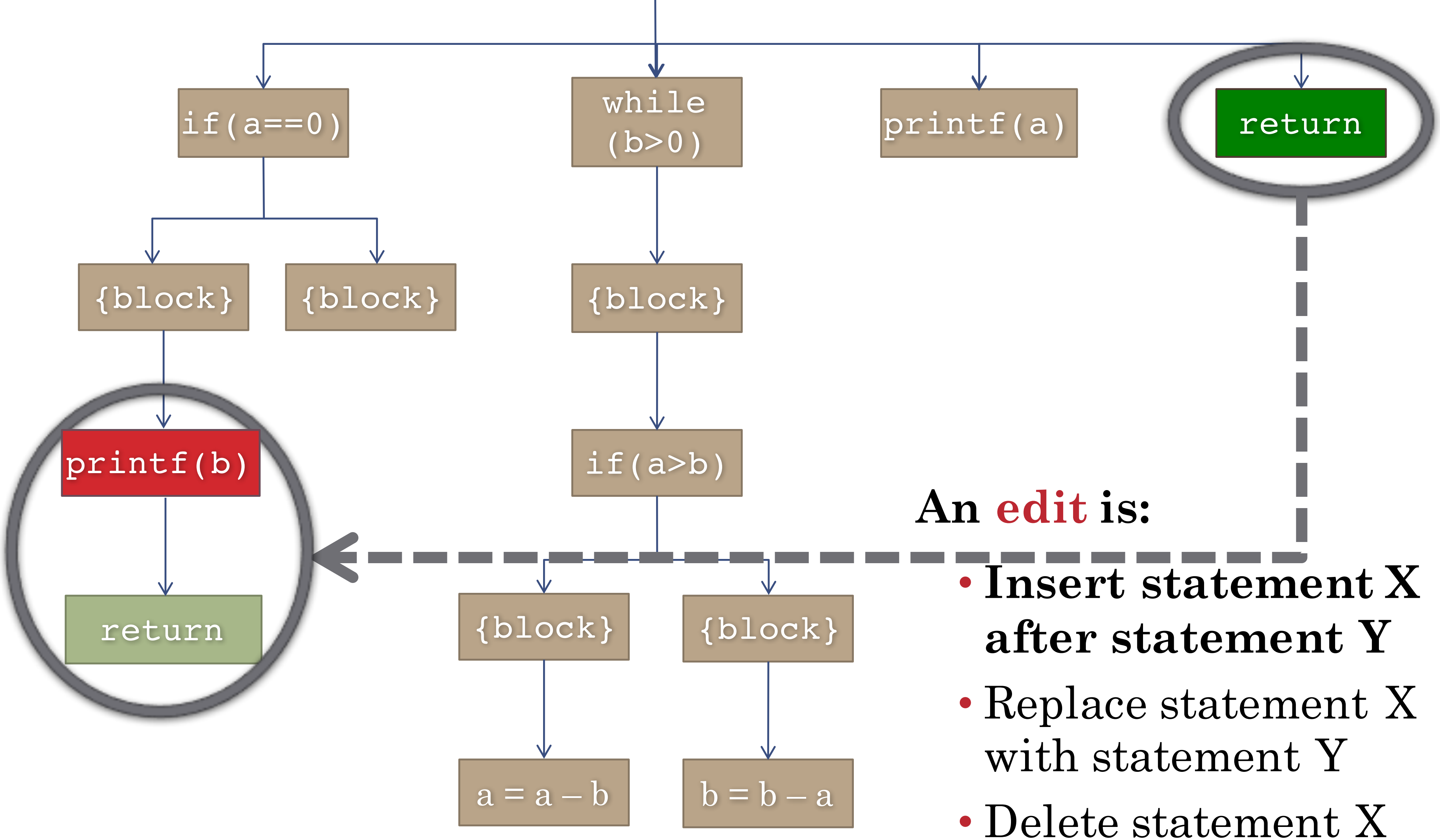
Ack: Claire Le Goues (CMU)

Input:



Ack: Claire Le Goues (CMU)

Input: 



An **edit** is:

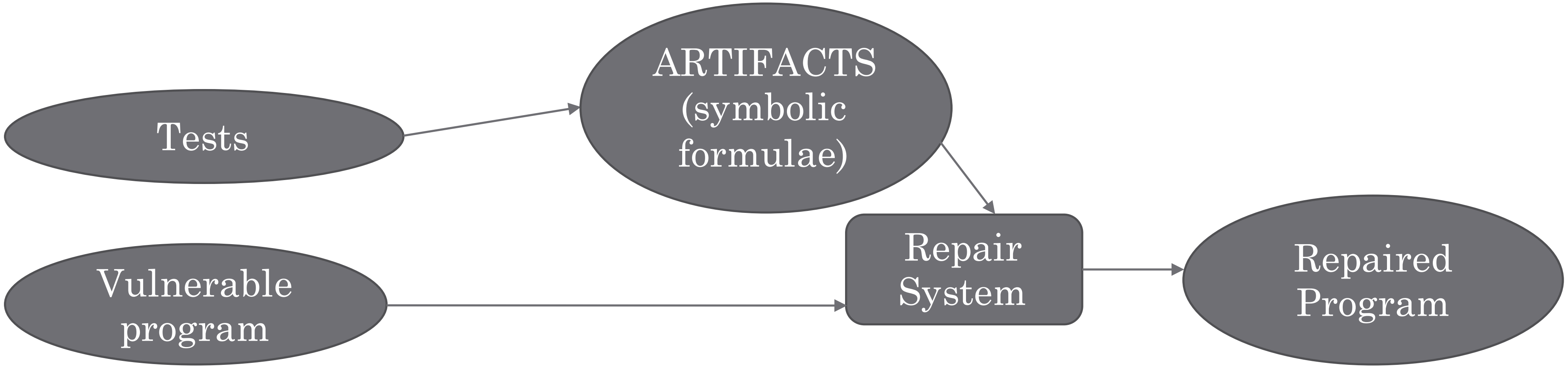
- **Insert statement X after statement Y**
- Replace statement X with statement Y
- Delete statement X

Ack: Claire Le Goues (CMU)

Over-fitting in Repair

Avoid generating programs like

if (input1) return output1
else if (input2) return output2
else if (input3) return output3
….



Generalize beyond the provided tests using symbolic reasoning.

Comparison

Syntactic Program Repair

Syntax-based Schematic
for e in Search-space{
    Validate e against Tests
}

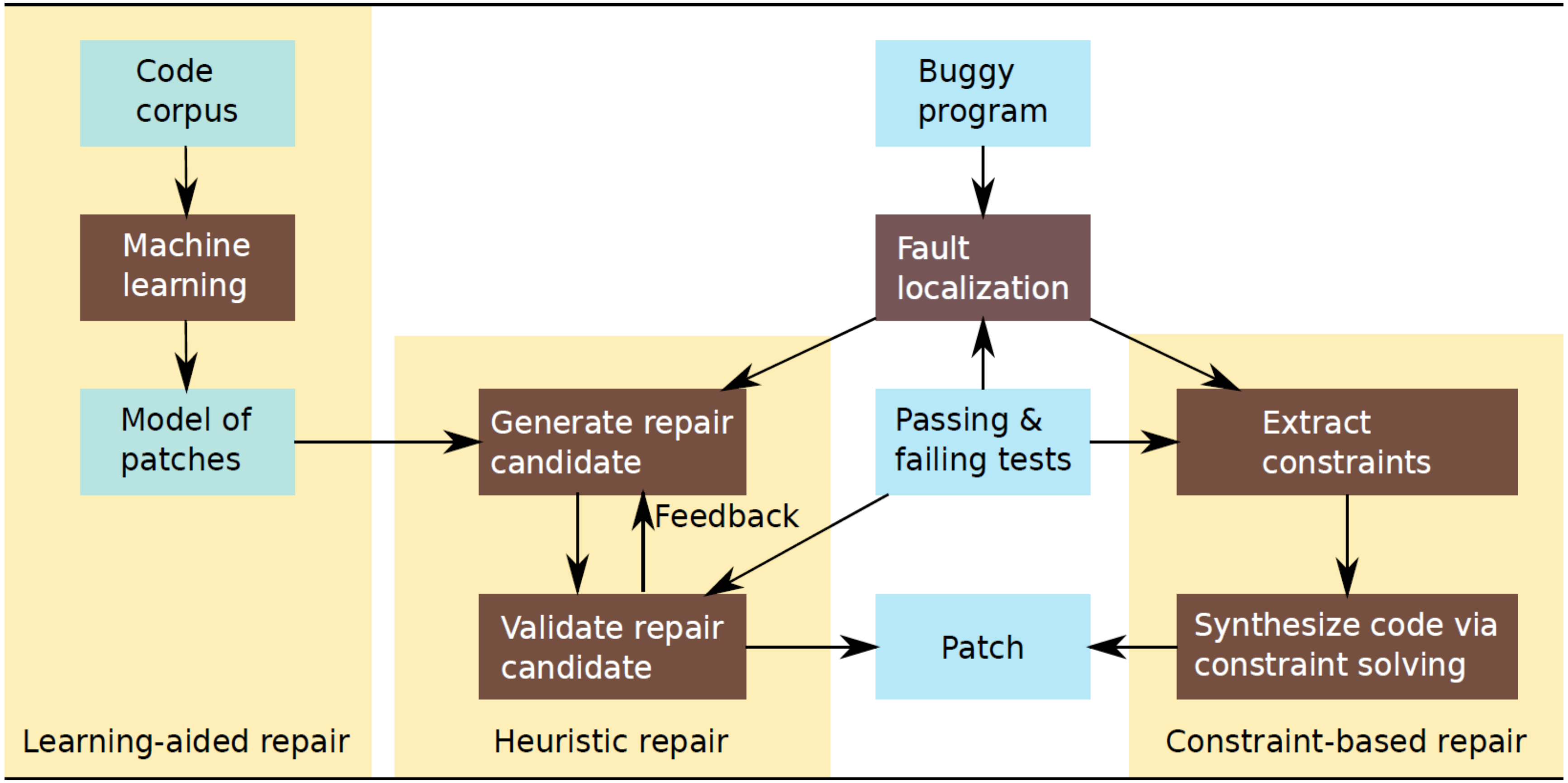
1. Where to fix, which line?
2. Generate patches in the candidate line
3. Validate the candidate patches against correctness criterion.

Semantic Program Repair

Semantics-based Schematic
for t in Tests {
    generate repair constraint Ψ_t
}
Synthesize e from $\wedge_t \Psi_t$

1. Where to fix, which line(s)?
2. What values should be returned by those lines, e.g. <inp ==1, ret== 0>
3. What are the expressions which will return such values?

State-of-the-art



Ack: Figure by Le Goues(CMU), Pradel (Darmstadt), Roychoudhury (NUS)

```
1 int triangle(int a, int b, int c){
2     if (a <= 0 || b <= 0 || c <= 0)
3         return INVALID;
4     if (a == b && b == c)
5         return EQUILATERAL;
6     if (a == b || b != c)  // bug!
7         return ISOSCELES;
8 return SCALENE;
9 }
```

Test id	a	b	c	oracle	Pass
1	-1	-1	-1	INVALID	pass
2	1	1	1	EQUILATERAL	pass
3	2	2	3	ISOSCELES	pass
4	2	3	2	ISOSCELES	fail
5	3	2	2	ISOSCELES	fail
6	2	3	4	SCALENES	fail

Correct fix
(a == b || b == c || a== c)

Traverse all *mutations* of line 6, and check

Hard to generate correct fix since a==c never appears elsewhere in the program.

OR

Generate the constraint

$f(2,2,3) \wedge f(2,3,2) \wedge f(3,2,2) \wedge \neg f(2,3,4)$

And get the solution

f(a,b,c) = `(a == b || b == c || a== c)`

Semantic Program Repair

Prof. Abhik Roychoudhury

National University of Singapore

Challenge 1: Search Space Explosion

Buggy program

```
scanf("%d", &x);
int t = x - 1;
if (t > 0) printf("1");
else printf("0");
```

Test

P(1) ⟶ 1

Failing test

Huge search space of candidate patches

x -1 ⟶ x - 2

x- 1 ⟶ x + 1

…

Challenge 2: Overfitting

Buggy program

```
scanf("%d", &x);
int t = x - 1;
if (t > 0) printf("1");
else printf("0");
```

Test

P(1) ⟶ 1

Failing test

Huge space of plausible patches

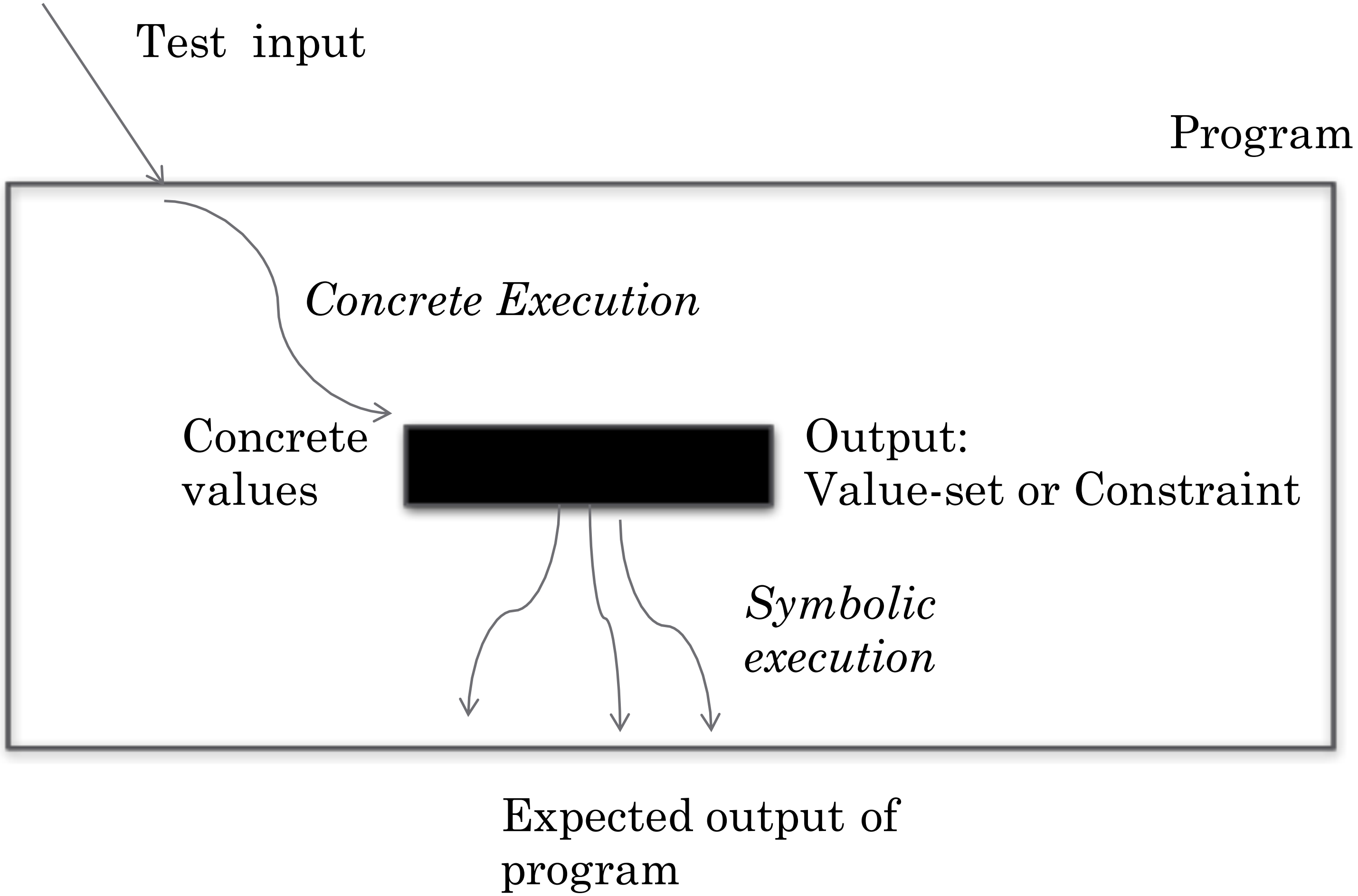
x -1 ⟶ 1

x- 1 ⟶ x

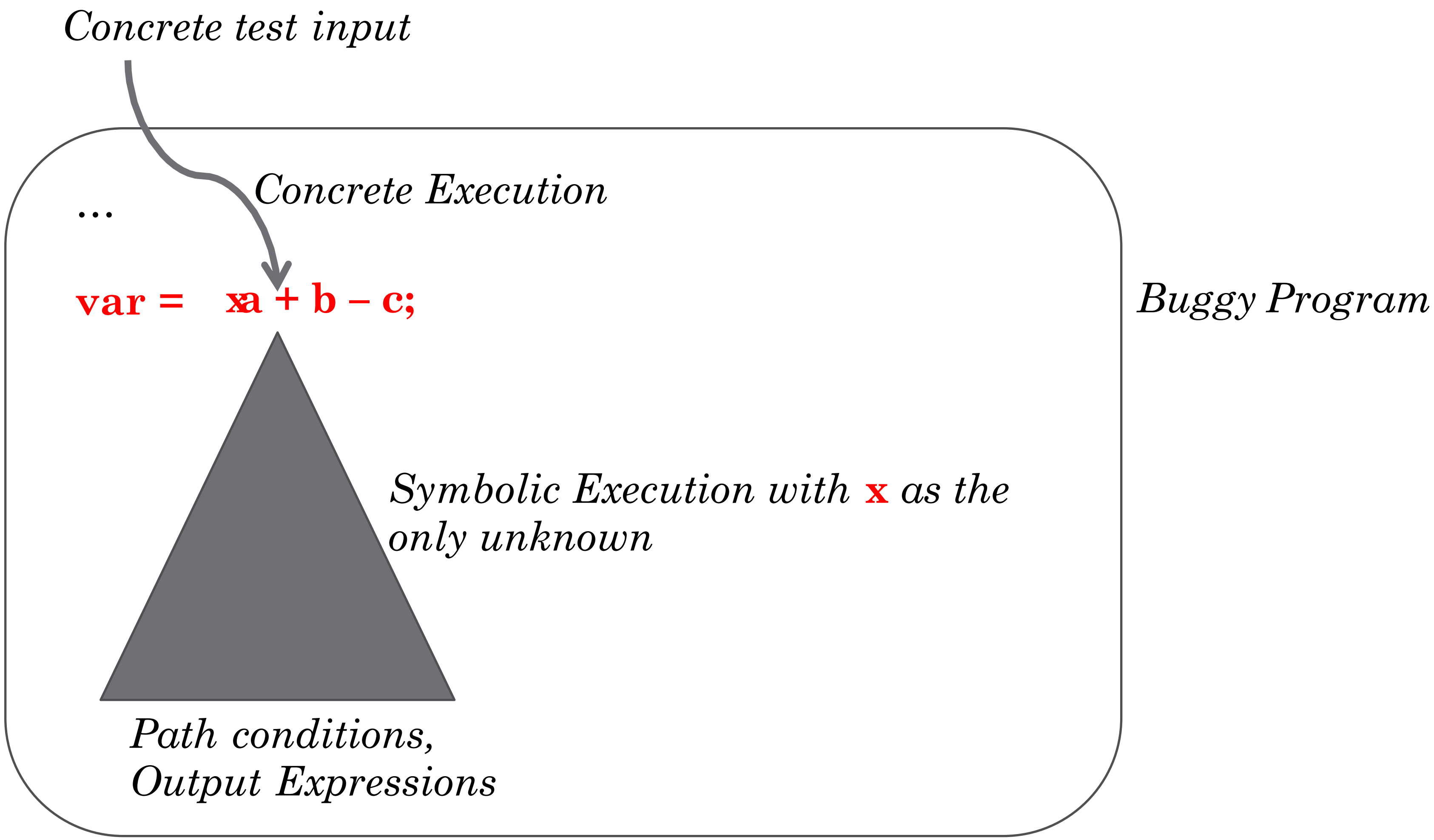
x- 1 ⟶ x + 1

…

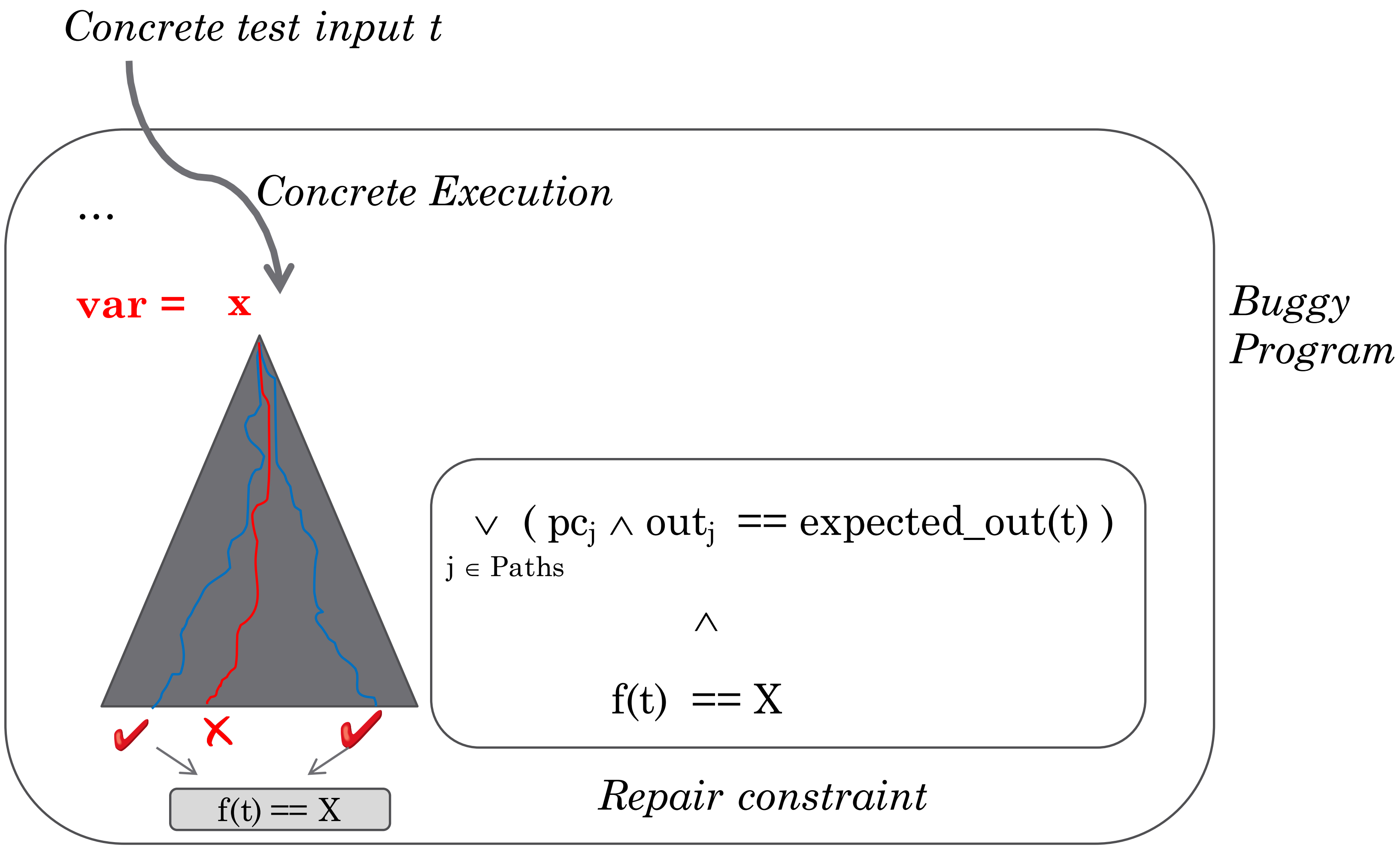
Specification Inference



What it should have been



What it should have been



Example

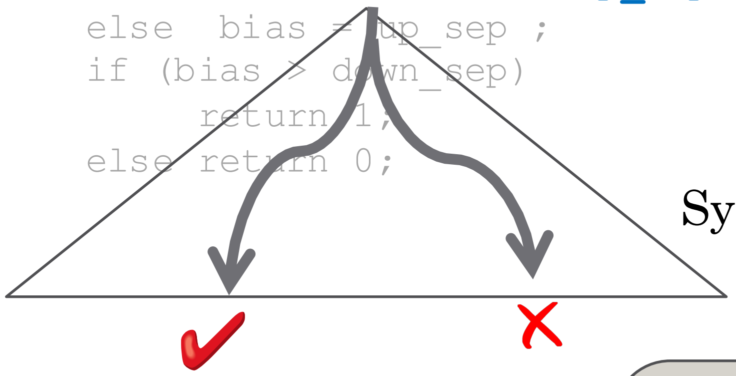
```c
int is_upward( int inhibit, int up_sep, int down_sep){
        int bias;
        if (inhibit)
            bias = down_sep; //   bias= up_sep + 100
        else  bias = up_sep ;
        if (bias > down_sep)
             return 1;
        else return 0;
}
```

inhibit	up_sep	down_sep	**Observed output**	**Expected Output**	**Result**
1	0	100	0	0	pass
1	***11***	***110***	***0***	***1***	***fail***
0	100	50	1	1	pass
1	***-20***	***60***	***0***	***1***	***fail***
0	0	10	0	0	pass

Example

Inhibit == 1	up_sep == 11	down_sep == 110

```
int is_upward( int inhibit, int up_sep, int down_sep){
        int bias;
        if (inhibit)
            bias = f(inhibit, up_sep, down_sep)   // X
        else  bias = up_sep ;
        if (bias > down_sep)
            return 1;
        else return 0;
}
```



Symbolic Execution

$$\Big((X > 110 \wedge 1 == 1) \vee (X \leq 110 \wedge 0 == 1) \Big) \wedge f(1,11,110) == X$$

$$\bigvee_{j \in \text{Paths}} \left(pc_j \wedge out_j == expected\_out(t) \right) \wedge f(t) == X$$

Repair constraint

What it should have been

Inhibit == 1	up_sep == 11	down_sep == 110

```
int is_upward( int inhibit, int up_sep, int down_sep){
        int bias;
        if (inhibit)
            bias = f(inhibit, up_sep, down_sep)
        else  bias = up_sep ;
        if (bias > down_sep)
            return 1;
        else return 0;
}
```

Symbolic Execution

f(1,11,110) > 110 ✔ ✘

Fix the suspect

- Accumulated constraints
  - $f(1,11,\ 110) > 110 \wedge$
  - $f(1,0,100) \leq 100 \wedge$
  - …
- Find a f satisfying this constraint
  - By fixing the set of operators appearing in f
- Candidate methods
  - Search over the space of expressions
  - Program synthesis with fixed set of operators
    - More efficient!!
- Generated fix
  - `f(inhibit,up_sep,down_sep) = up_sep + 100`

Function synthesis

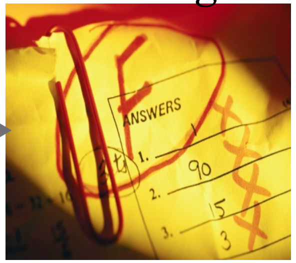
- Instead of solving

Repair Constraint:
$f(1,11,110) > 110 \wedge f(1,0,100) \leq 100 \wedge f(1,-20,60) > 60$

- Select primitive components to be used by the synthesized program based on complexity
- **Look** for a program that uses only these primitive components and satisfy the repair constraint
  - Done via another constraint solving problem – pgm. synthesis
- **Solving the repair constraint is the key, not how it is solved**
- Enumerate expressions over a given set of components / operators
  - Enforce axioms of the operators
  - If candidate repair contains a constant, solve using SMT

Patch as minimal change

Failing tests



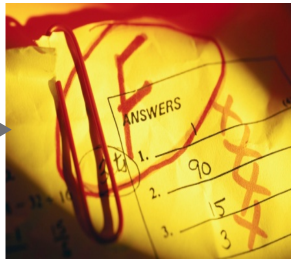
Debugging

DSE

Synthesis

MaxSMT solver

Failing tests



Conjure a function which represents minimal change to buggy program.

Example

```
if (x > y)
  if (x > z)
      out  =10;
    else
      out = 20;
else
      out = 30;
return out;
```

Test cases:
all possible orderings of x,y,z

SemFix →

```
if (x > y)
  if (x > z)
      out  =10;
    else
      out = 20;
else
      out = 30;
return ((x==y)? ((x==z)?10: 20)): out);
```

DirectFix →

```
if (x >= y)
  if (x >= z)
      out  =10;
    else
      out = 20;
else
      out = 30;
return out;
```

No fault localization

```
int foo(int x, int y){
   if (x > y)
      y = y + 1;
   else
      y = y – 1;
   return y + 2;
}
```

Test: foo(0,0) == 3?

$x = 0 \wedge y = 0 \wedge result = 3$

$\wedge$

(if (x1 > y1) then (y2 = y1 + 1) else (y2 = y1 – 1)
$\wedge$
(result = y2 + 2)
)

= **UNSAT**

Constraint = Whole Program

$x = 0 \wedge y = 0 \wedge result = 3$

$\wedge$

(if (x1 > y1) then (y2 = y1 + 1) else (y2 = y1 – 1)
  $\wedge$
  (result = y2 + 2)
)

= **UNSAT**

(if (x1 **>=** y1) then (y2 = y1 + 1) else (y2 = y1 – 1)
  $\wedge$
  (result = y2 + 2)
)

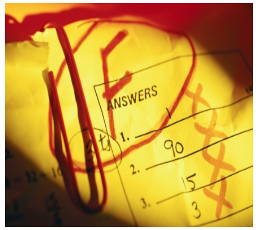
$\wedge$

$x = 0 \wedge y = 0 \wedge result = 3$

= **SAT**

Need Concise Constraints

Failing tests



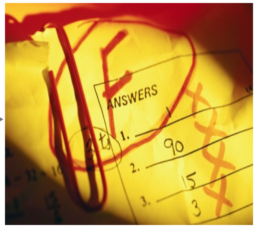
MaxSMT solver

Minimized Mutations for Repair

MaxSMT solver

Concise Semantics Signature

Failing tests



DSE

Angelic Values

Syntax-based Schematic

```
for  e in SearchSpace{
      Validate e against Tests
}
```

Semantics-based Schematic

```
for  t in Tests {
      generate repair constraint Ψt
}
Synthesize e from ∧tΨt
```

Instead of representing Ψ_t
as a SMT constraint represent it using **values**.

Value that is arbitrarily set during execution to a selected expression and that makes the program pass.
Can be found by solving path condition of failing test case (I, O):

$$pathcondition[\alpha] \wedge input = I \wedge output = O$$

Angelic Values

Buggy program

```
scanf("%d", &x);
int t = α ;
if (β) printf("1");
else printf("0");
```

Test

P(1) ⟶ 1

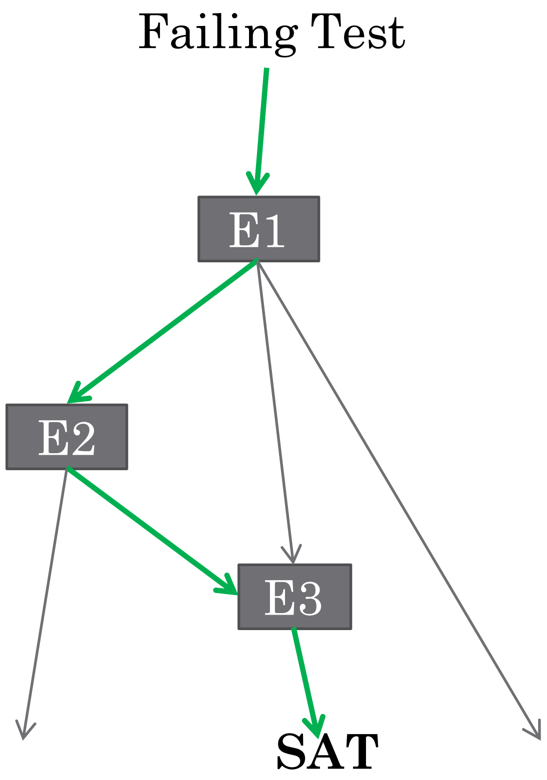
Extract value based specification

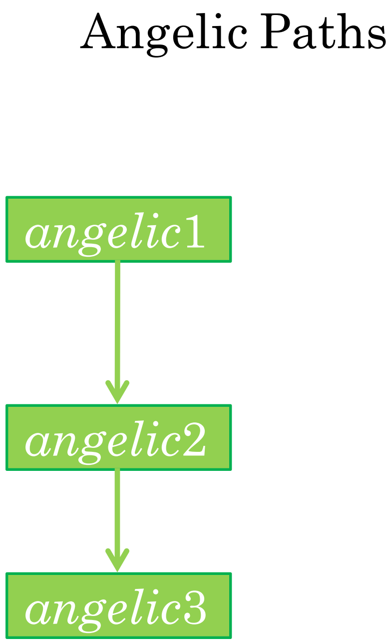
$\langle \alpha = 2, \sigma = \{ x \rightarrow 1 \} \rangle$ $\langle \beta = \text{true}, \sigma = \{ x \rightarrow 1, t \rightarrow 2 \} \rangle$

Angelic forest: Patch synthesis specification based on

Angelic values $\{\langle \text{Symbolic Variable name, Constant, State} \rangle\}_{\textit{Paths,Tests}}$

Angelic Forest

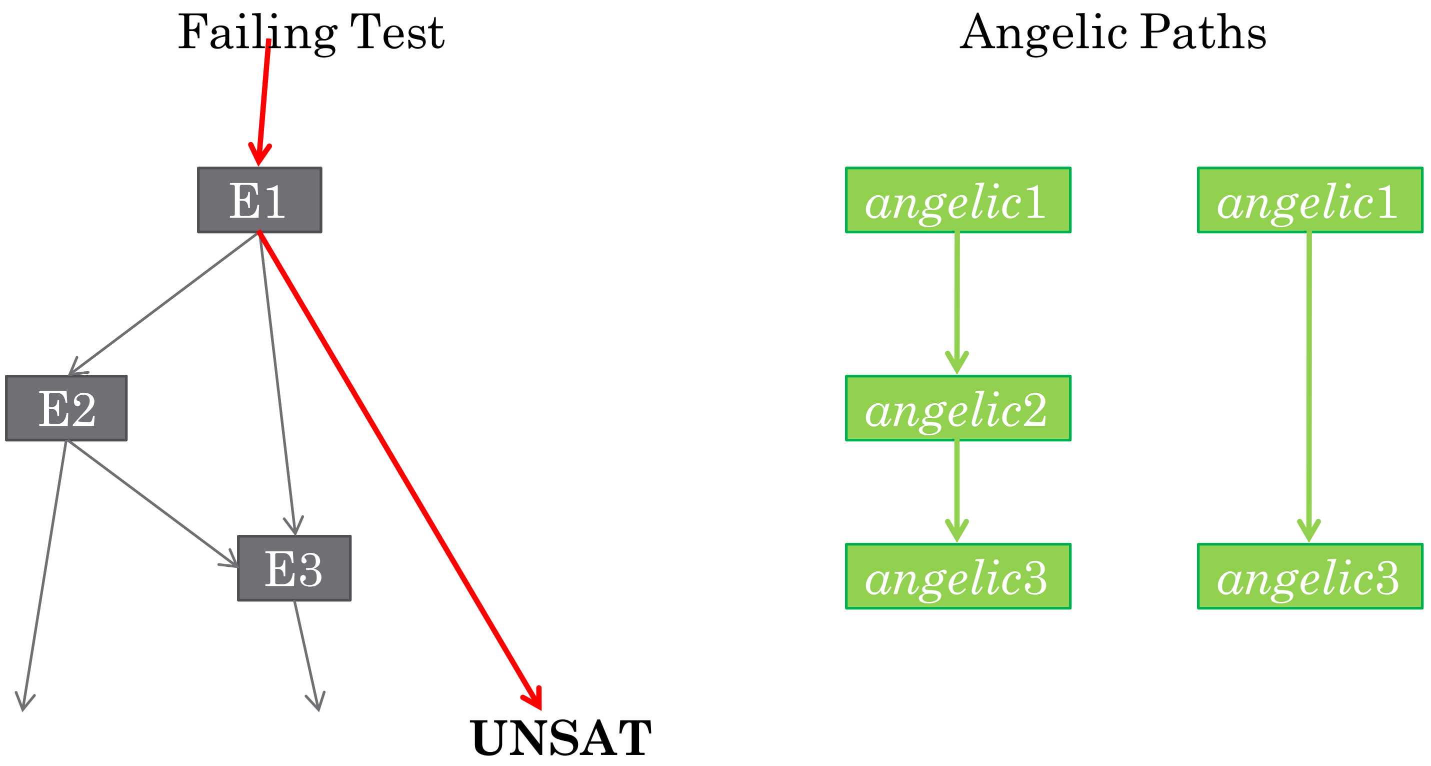




Angelic forest: Patch synthesis specification based on

Angelic values $\{\langle \text{Symbolic Variable name, Constant, State} \rangle\}_{Paths,Tests}$

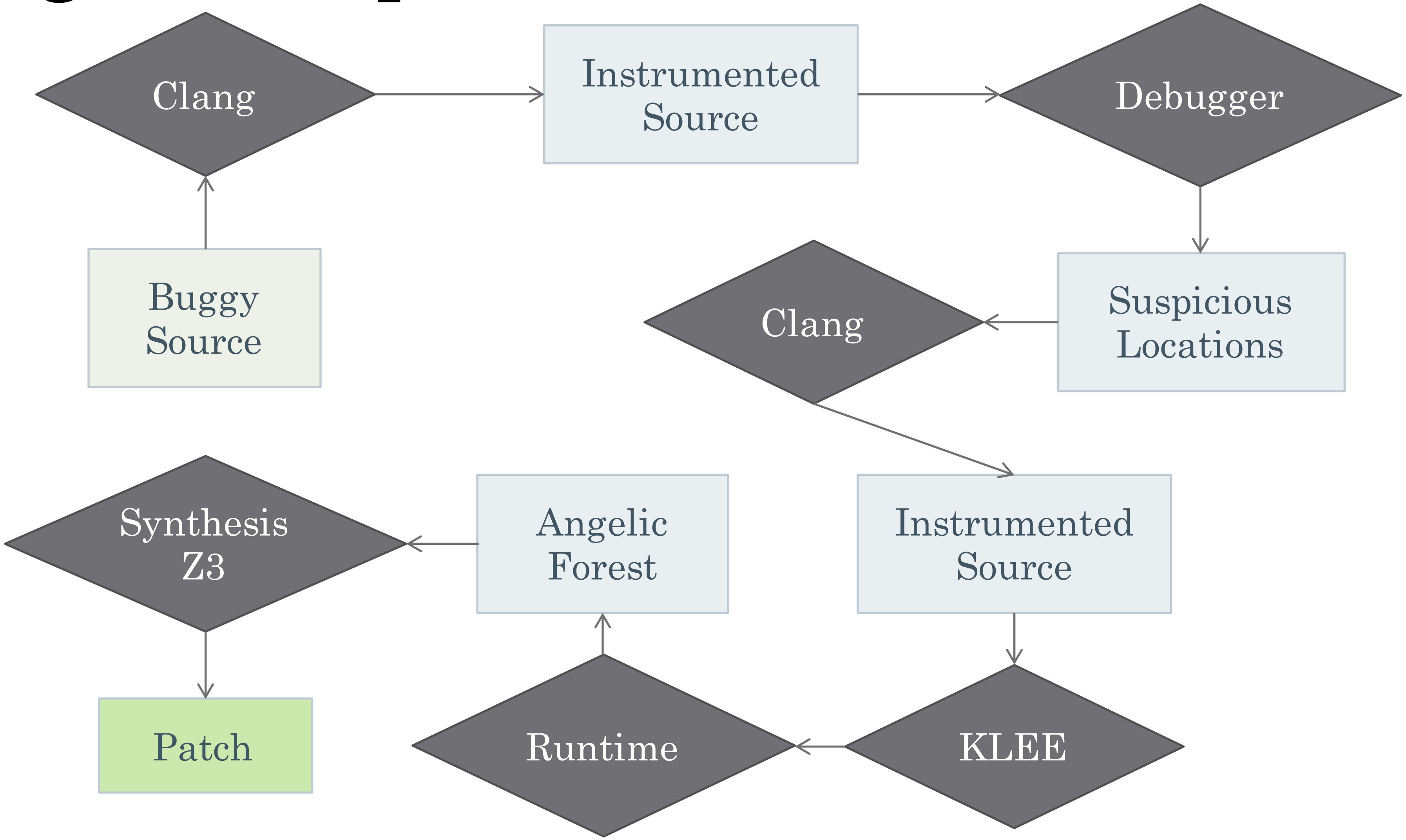
Angelic Forest



Angelic forest: Patch synthesis specification based on

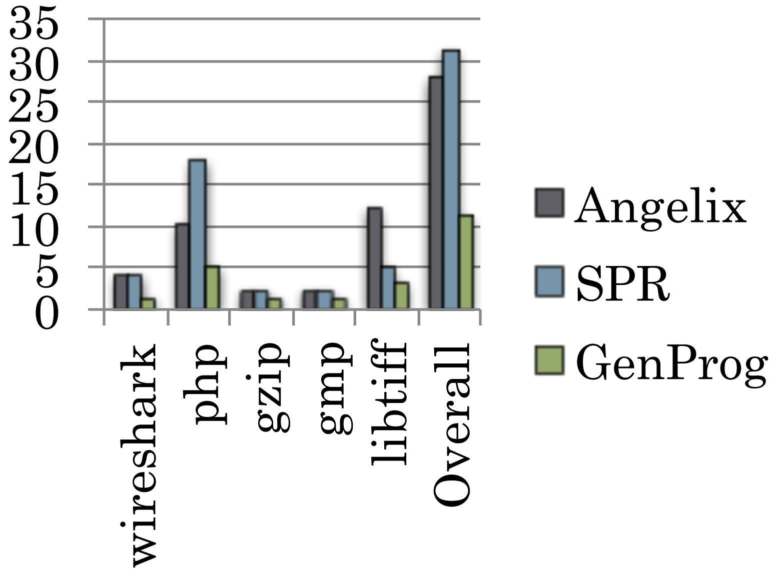
Angelic values $\{\langle \text{Symbolic Variable name, Constant, State} \rangle\}_{Paths,Tests}$

Angelix Implementation



Results

Subject	LoC
wireshark	2814K
php	1046K
gzip	491K
gmp	145K
libtiff	77K



	#Fixes	Del	Del, Per
Angelix	28	5	18%
SPR	31	13	42%

Multiline Results

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

The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).
--- *Source: heartbleed.com*

```
if ( hbtype == TLS1 HB REQUEST) {
    . . .
    memcpy (bp , pl , payload ) ;
    . . .
}
```

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

```
if ( hbtype == TLS1 HB REQUEST
    && payload + 18 < s->s3->rrec.length) {
    . . .
}
```

(b) A fix generated automatically

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
        return 0;
. . .
if ( hbtype == TLS1_HB_REQUEST) {
    . . .
}
else if (hbtype == TLS1_HB_RESPONSE){
    . . .
}
return 0;
```

(c) The developer-provided repair

Research Issues in Program Repair

- [**OLD**] *Large search space* of candidate patches for general-purpose repair tools.
- ->. ***What should I use?***
- ***-> Which search frameworks could we use?***
- ***-> Syntactic Program Repair***

- **[NEW]** Weak description of intended behavior / *correctness criterion* e.g. tests
- -> ***Overfitting of a patch candidate to tests?***
- -> ***Extract specification from test executions to reduce overfitting.***
- ***->. Do so, while still navigating the search space***
- ***-> Semantic Program Repair***

Spec. from reference implementation

User-define condition: length = 3 & a[0] < a[1] < a[2]

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

(a) Correct linear search

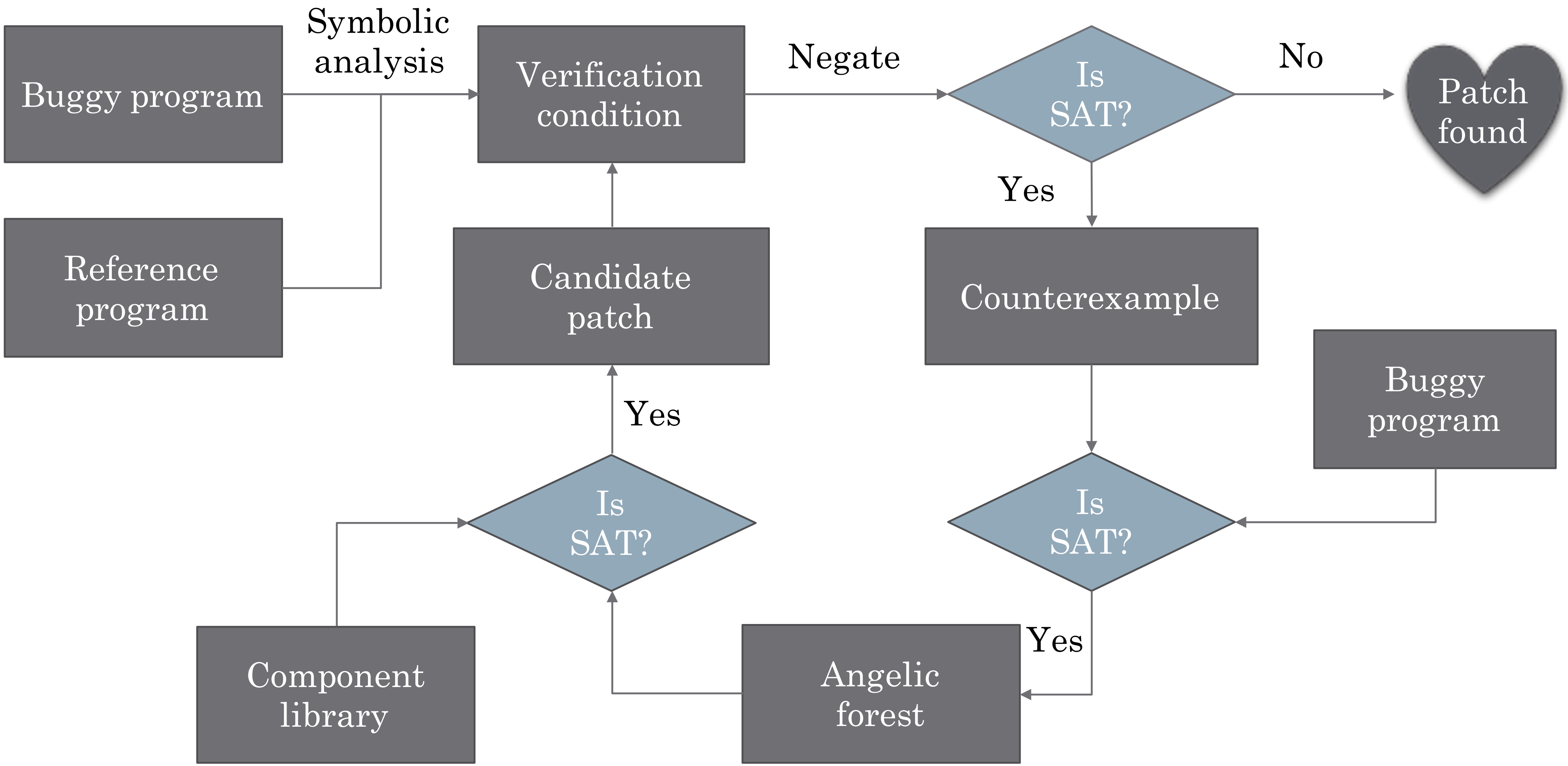
```
int search(int x, int a[], int length) {
  int L = 0;
  int R = length-1;
  do {
    int m = (L+R)/2;
    if (x == a[m]) {
      return m;
    } else if (x < a[m]) { // bug fix: x > a[m]
      L = m+1;
    } else {
      R = m-1;
    }
  } while (I
  return -1;
}
```

(b) Buggy binary search

Verification condition

Experiments on embedded Linux Busybox

SemGraft


Buggy program
Symbolic analysis
Reference program
Verification condition
Negate
Is SAT?
No
Patch found
Yes
Counterexample
Buggy program
Is SAT?
Yes
Angelic forest
Candidate patch
Yes
Is SAT?
Component library

SemGraft Results

GNU Coreutils as reference

Program	Commit	Bug	Angelix	SemGraft
sed	c35545a	Handle empty match	Correct	Correct
seq	f7d1c59	Wrong output	Correct	Correct
sed	7666fa1	Wrong output	Incorrect	Correct
sort	d1ed3e6	Wrong output	Incorrect	Correct
seq	d86d20b	Don't accepts 0	Incorrect	Correct
sed	3a9365e	Handle s///	Incorrect	Correct

Linux Busybox as reference

Program	Commit	Bug	Angelix	SemGraft
mkdir	f7d1c59	Segmentation fault	Incorrect	Correct
mkfifo	cdb1682	Segmentation fault	Incorrect	Correct
mknod	cdb1682	Segmentation fault	Incorrect	Correct
copy	f3653f0	Failed to copy a file	Correct	Correct
md5sum	739cf4e	Segmentation fault	Correct	Correct
cut	6f374d7	Wrong output	Incorrect	Correct

GNU Coreutils Cut

GNU Coreutils wrongly interprets the command -b 2-,3- as -b 3- (extract input bytes starting from the third byte):

```
echo -ne ’1234 ’ | cut -b 2-,3-
34
```

instead of -b 2- (extract input bytes starting from the second byte):

```
echo -ne ’1234 ’ | cut -b 2-,3-
234
```

Developer tests:

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

GNU Coreutils cut

Automatic patch based on developer tests

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

Developer patch

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

Parameterized test to improve automated repair and apply SemGraft

```
echo -ne '1234 ' | cut -b σ-,β-
```

Recap: Comparison

Syntactic Program Repair

Syntax-based Schematic
for e in Search-space{
Validate e against Tests
}

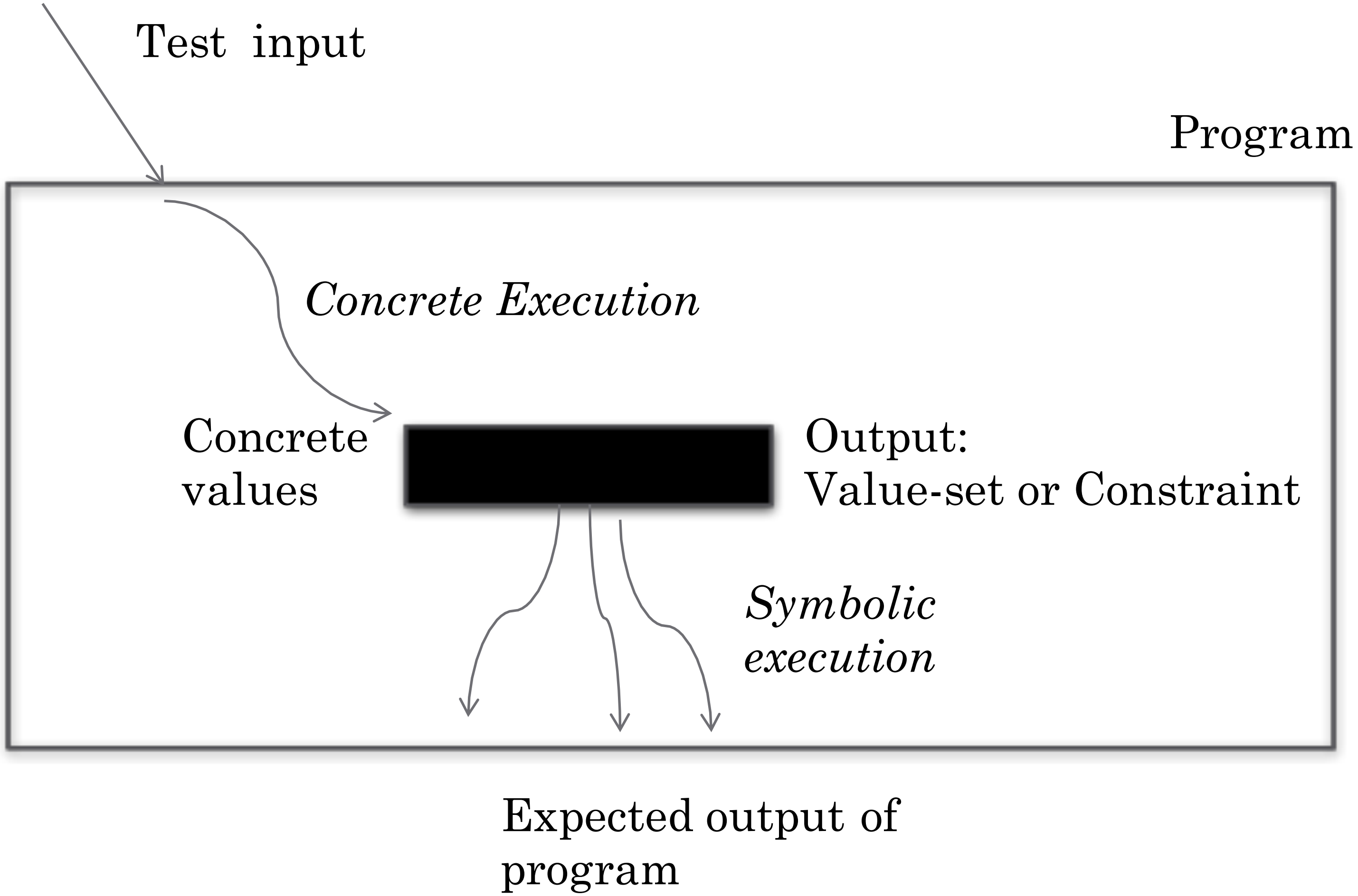
1. Where to fix, which line?
2. Generate patches in the candidate line
3. Validate the candidate patches against correctness criterion.

Semantic Program Repair

Semantics-based Schematic
for t in Tests {
generate repair constraint Ψ_t
}
Synthesize e from $\wedge_t \Psi_t$

1. Where to fix, which line(s)?
2. What values should be returned by those lines, e.g. <inp ==1, ret== 0>
3. What are the expressions which will return such values?

Specification Inference



Revisiting Program Synthesis

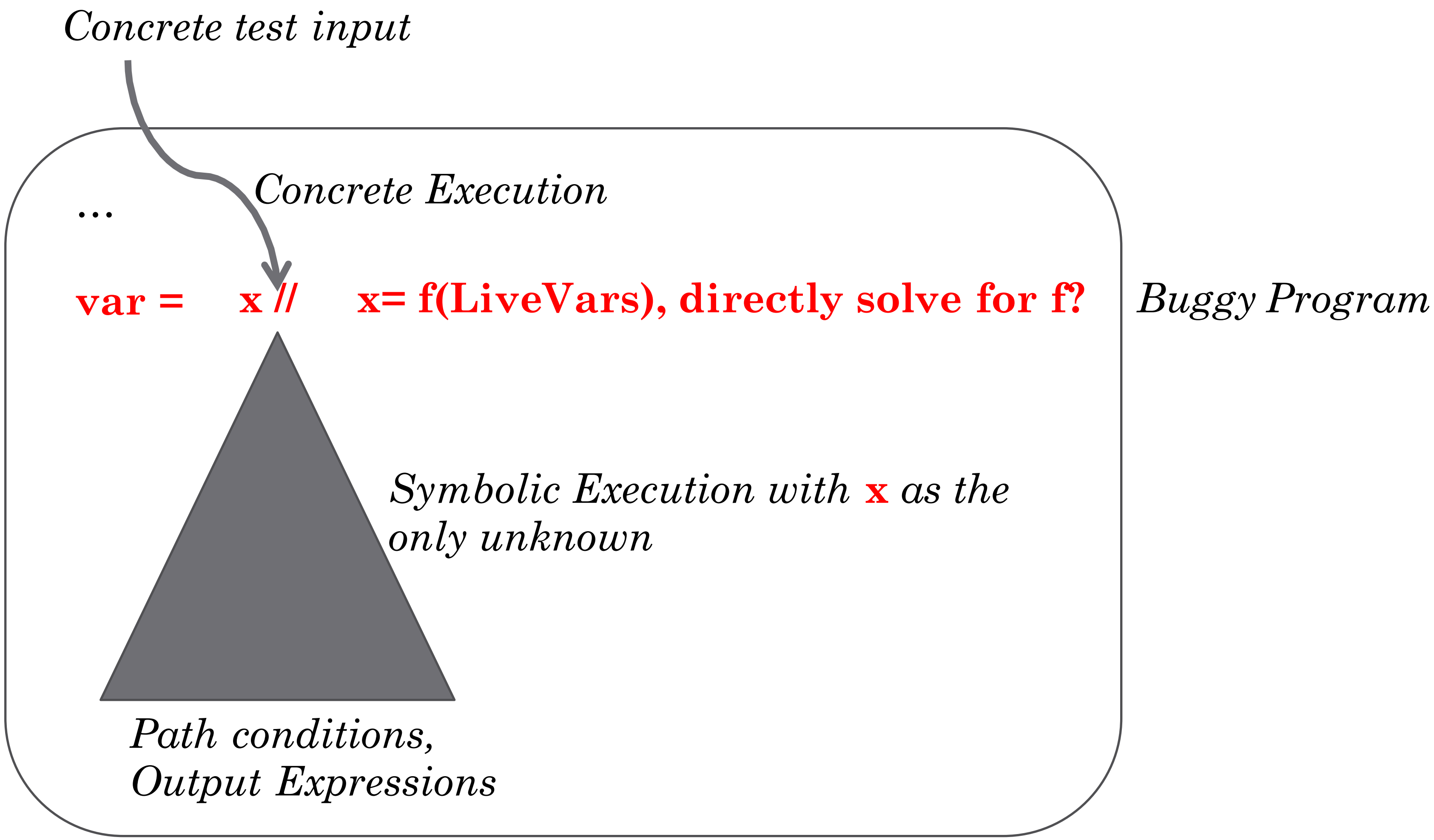
From input-output examples

$$\exists p \in P. \bigwedge_{(\sigma,o)\in T} [\![p]\!]_\sigma = o$$

where P is a set of well-formed programs to choose from (candidate patches)
T is a given set of tests

Program Repair involves solving for such program fragments from input-output examples or input-output constraints, amounting to second order reasoning.

Repair via 2nd order reasoning



First order Symbolic Execution

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

Symbolic Inputs α, β, γ

search((int[]){α,β,γ}, 3, positive)

First order Symbolic Execution

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

Symbolic Execution results of `search((int[]){α,β,γ}, 3, positive)`

Path Condition	Input	Output
$\alpha > 0$	{1,0,0}	0
$\alpha \leq 0 \wedge \beta > 0$	{0,1,0}	1
$\alpha \leq 0 \wedge \beta \leq 0$	{0,0,1}	2
$\alpha \leq 0 \wedge \beta \leq 0 \wedge \gamma \leq 0$	{0,0,0}	3

(Our) Second order reasoning

- Allow for existentially quantified second order variables.
- Restrict their interpretation to a language e.g. linear integer arithmetic

$$Term = Var \mid Constant \mid Term + Term \mid Term - Term \mid Constant * Term$$

- *SAT*
  - $\rho(0) > 0 \wedge \rho(1) \leq 0$
  - Satisfying solution $\rho = \lambda x.\ 1 - x$
- *UNSAT*
  - $\rho(0) > 0 \wedge \rho(1) \leq 0 \wedge \rho(2) > 0$
  - All functions in LIA are monotonic.

Second order Symbolic Execution

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

Symbolic Execution results of `search((int[]){0,1,2}, 3, ρ)`

Path Condition	ρ	Output
$\rho(0)$	$\lambda x.\ true$	0
$\neg\ \rho(0) \wedge \rho(1)$	$\lambda x.\ x>0$	1
$\neg\ \rho(0) \wedge \neg\ \rho(1) \wedge \rho(2)$	$\lambda x.\ x>1$	2
$\neg\ \rho(0) \wedge \neg\ \rho(1) \wedge \neg\ \rho(2)$	$\lambda x.\ false$	3

Syntactic Program Repair

Buggy Program:

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

Sample Test:

P(5) → "1110000000" expected "1111111000"

Generate and validate based repair tools:

Enumerate and test $P[x - i \rightarrow x + i]$, $P[x - i \rightarrow x - 1]$, ...

First order Semantic Program Repair

Buggy Program:

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

Sample Test:

P(5) → "1110000000" expected "1111111000"

Synthesis Specification:

$\exists\, e \in \text{Term}.\ \bigvee_i \pi_i\,[\alpha \to e] \wedge \text{output} = \text{expected}$

Background theory LIA

Second order Program Repair

Buggy Program:

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

Sample Test:

P(5) → "1110000000" expected "1111111000"

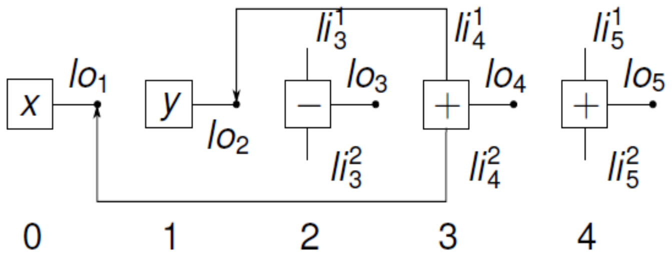
Synthesis Specification:

$\exists \rho.\ \bigvee_i \pi_i \wedge \text{output} = \text{expected}$

Solve for ρ directly

Term = *Var* | *Constant* | *Term* + *Term* | *Term* – *Term* | *Constant* * *Term*

(Old)Encoding for synthesis in 1st order



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

$$\phi_{range} := \bigwedge_{i \in [1,C]} \left(0 \le lo_i < C \ \wedge \bigwedge_{j \in [1,N_i]} 0 \le li_i^j < C \right)$$

$$\phi_{cons} := \bigwedge_{i,j \in [1,C], i \neq j} lo_i \neq lo_j$$

$$\phi_{acyc} := \bigwedge_{i \in [1,C], j \in [1,N_i]} lo_i > li_i^j$$

$$\phi_{conn} := \bigwedge_{i,j \in [1,C], k \in [1..N_i]} lo_i = li_j^k \Rightarrow \mathtt{out}_i = \mathtt{in}_j^k$$

(Recap) Second order reasoning

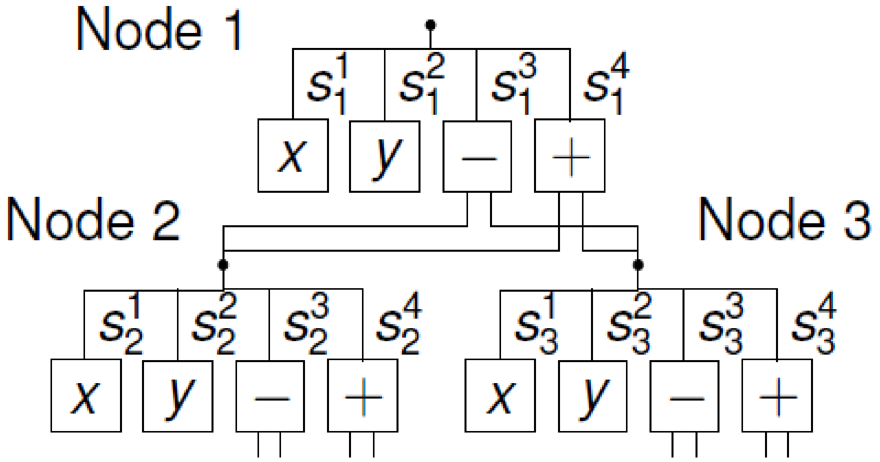
- **Allow for existentially quantified second order variables.**
- **Restrict their interpretation to a language e.g. linear integer arithmetic**

$$Term = Var \mid Constant \mid Term + Term \mid Term - Term \mid Constant * Term$$

- *Example SAT*
  - $\rho(0) > 0 \wedge \rho(1) \leq 0$
  - Satisfying solution $\rho = \lambda x.\ 1 - x$

Devise a propositional encoding to capture the set of interpretations

(New) Propositional Logic encoding



$$s_1^1 \mapsto x$$

$$s_1^3 \wedge s_2^1 \wedge s_3^2 \mapsto x - y$$

$$s_1^4 \wedge s_2^1 \mapsto \{x + T\}_{T \in Term}$$

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

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

Application in Repair: results

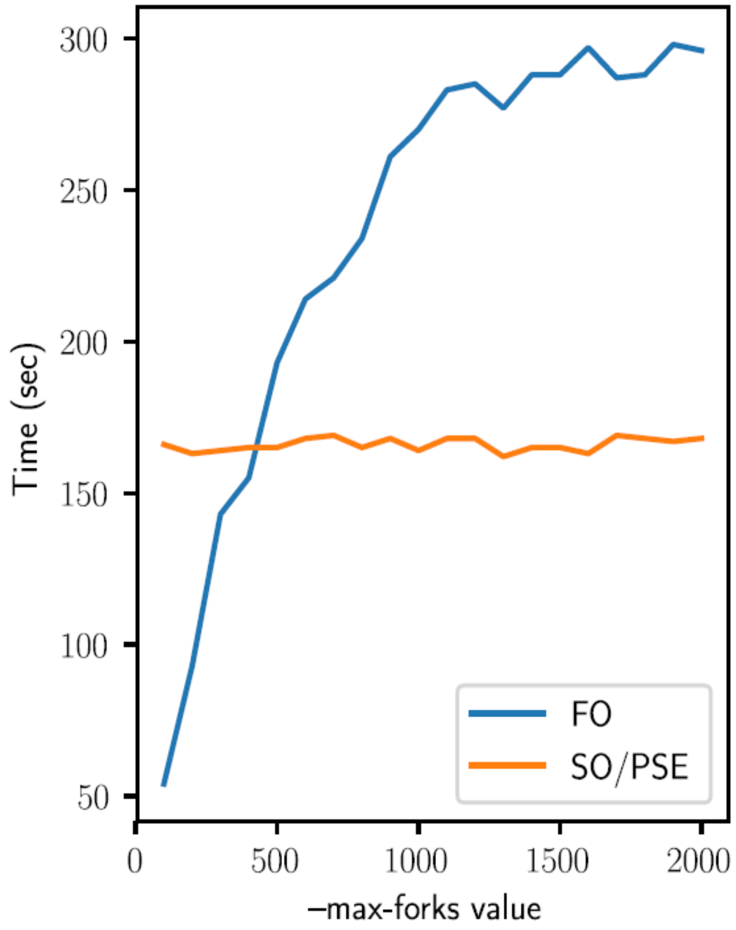
```
…   error_severity(1);
     return;
}
// ρ(ent->fts_info, ent->fts_errno, prev_depth)
else if  (ent->fts_info == FTSSLNONE){
     if  (symlink_loop(ent->fts_accpath))
…
```

`find` in GNU Coreutils

2000 paths in traditional first order Symbolic Execution

```
ρ1  :=(4 <= ent->fts_info)
ρ2  :=!(ent->fts_errno == prev_depth)
ρ3  :=((4 < ent->fts_info) && (prev_depth <= ent->fts_errno))
ρ4  :=!(0 == ent->fts_errno)
ρ5  :=((ent->fts_info == ent->fts_errno) || (9 <= prev_depth))
ρ6  :=(ent->fts_info == (7 + prev_depth))
ρ7  :=((prev_depth + ent->fts_errno) == (ent->fts_info - 6))
ρ8  :=((ent->fts_errno < prev_depth) || (6 == ent->fts_info))
ρ9  :=(ent->fts_info < (4 + ent->fts_errno))
ρ10 :=(ent->fts_info <= (ent->fts_errno + 6))
ρ11 :=!(ent->fts_info == 6)
ρ12 :=(0 <= prev_depth)
ρ13 :=!(4 < ent->fts_info)
ρ14 :=!(1 <= prev_depth)
ρ15 :=((ent->fts_errno < prev_depth) || (ent->fts_info <= 1))
ρ16 :=((ent->fts_errno < 32) || (prev_depth == ent->fts_info))
```

Comparison: 1st and 2nd order logic



Time taken by second order symbolic execution is independent of the maximum number of paths explored.

Other applications

- Modeling libraries for symbolic execution of application program.
  - Do not manually provide libraries for symbolic analysis.
  - Instead, they can be partially *synthesized*.

```
void main (int argc , char * argv [])
{
int a = atoi( argv [1]) ;
printf ("%d\n", 16 / a);
}
```

P(“4") → “4";
P(“16") → “1"

Tests

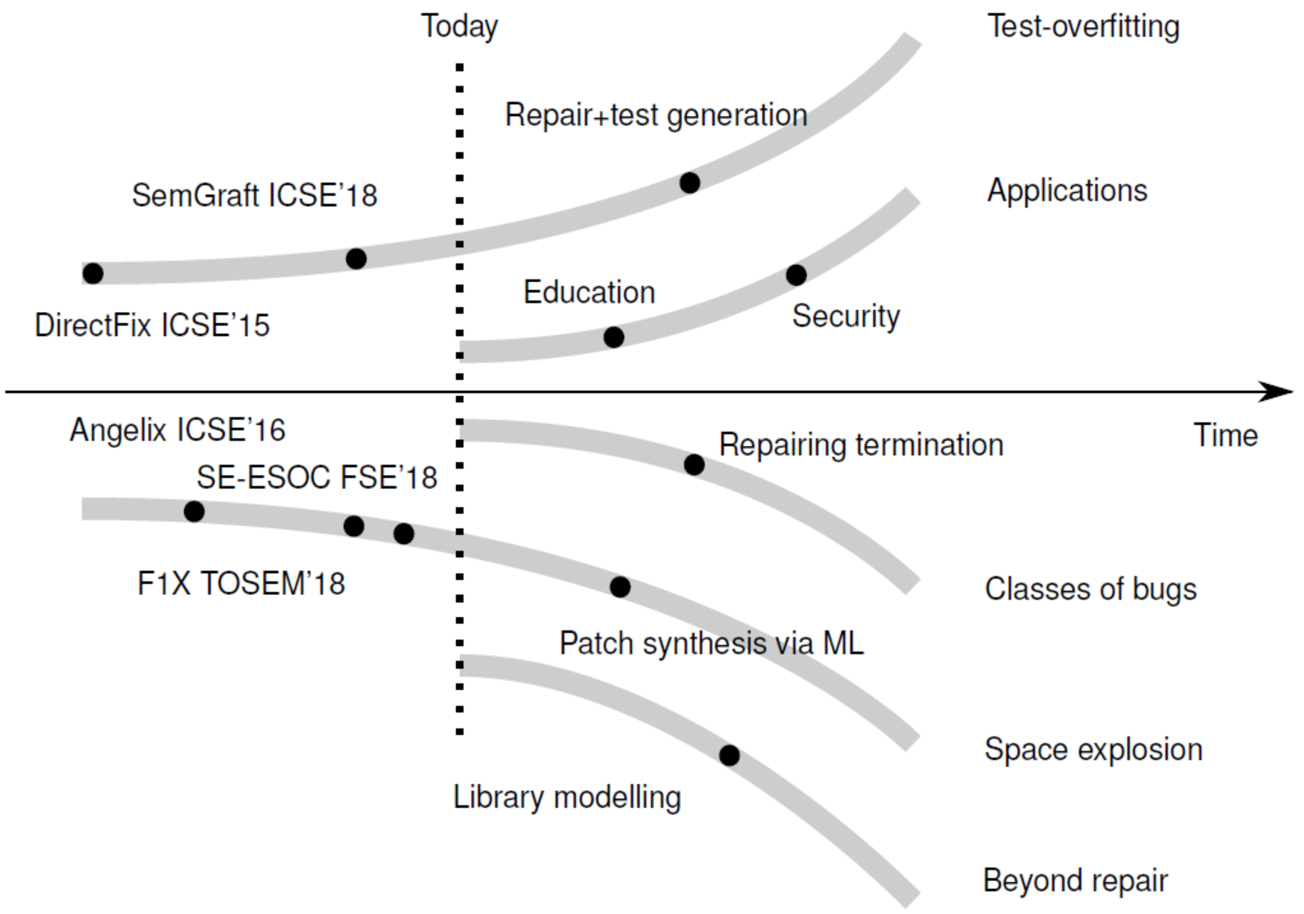
Second order reasoning

```
void atoi_sketch ( char *arr []) {
int acc;
for (i = 0; i < strlen (arr); i++)
acc = ρ(acc , arr[i]);
return acc;
}
```

$\rho = \lambda xy.\ 10\,x + y - 48$

Use this ρ and symbolic execution to find crashing inputs e.g. “0”

Future work in Semantic Repair

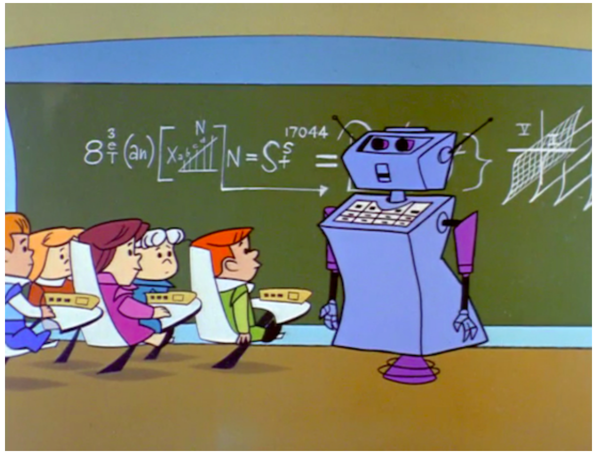


Briefly:
Novel applications outside security

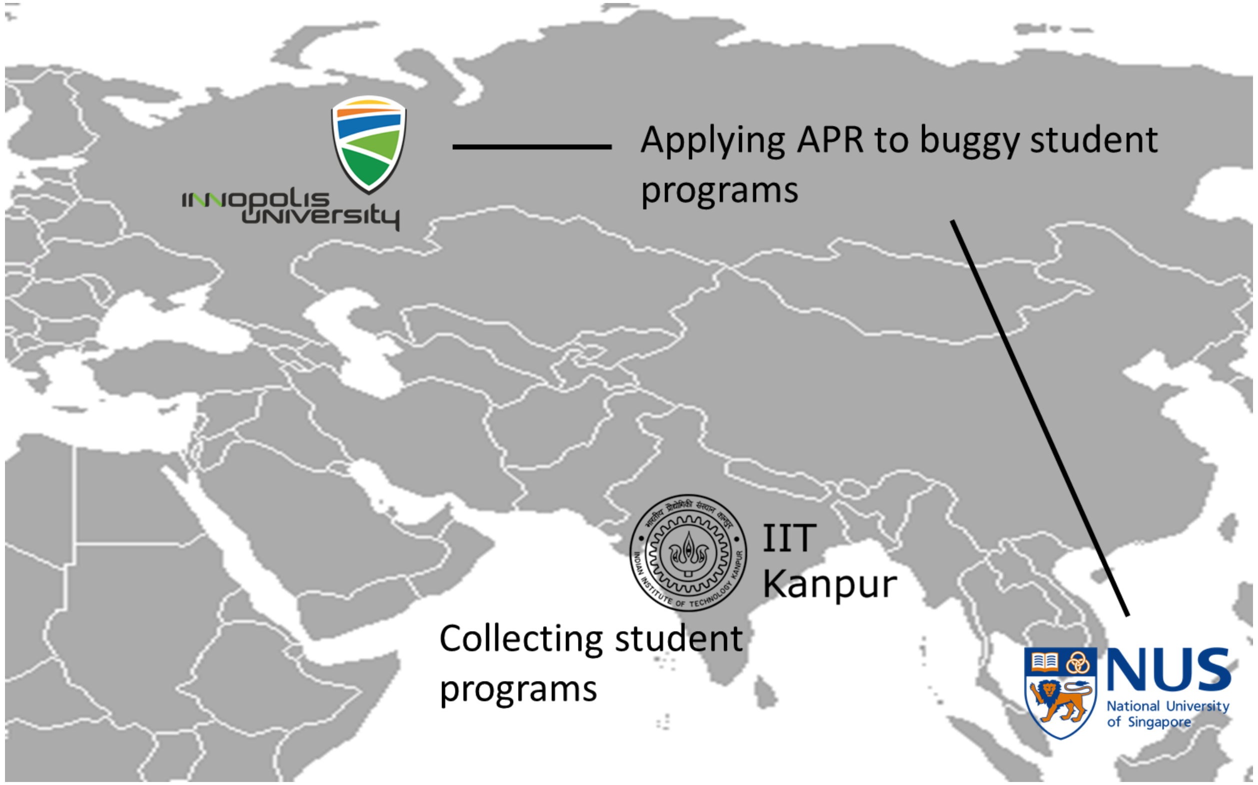
Use program repair in intelligent tutoring systems to give the students' individual attention.

Study in IIT-Kanpur (FSE17)

Application in Education



Intelligent tutoring

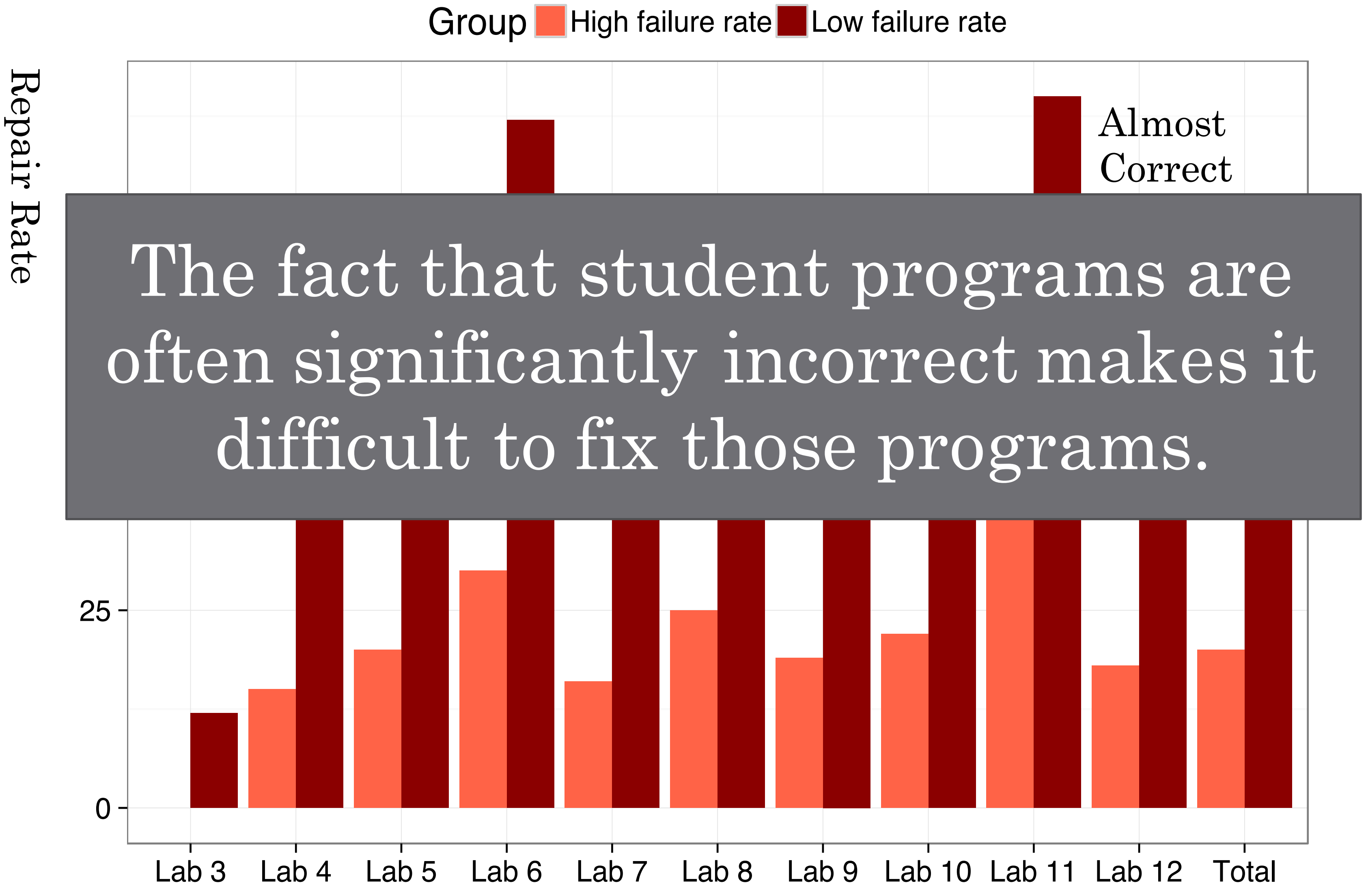


Dataset Preparation

- Lab: Programming assignments

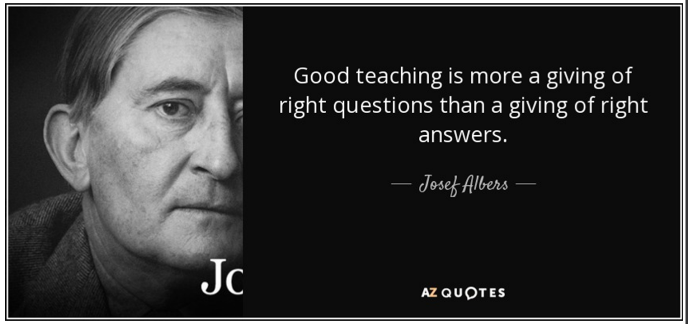
Lab	# Prog	Topic
Lab 3	63	Simple Expressions, printf, scanf
Lab 4	117	Conditionals
Lab 5	82	Loops, Nested Loops
Lab 6	79	Integer Arrays
Lab 7	71	Character Arrays (Strings) and Functions
Lab 8	33	Multi-dimensional Arrays (Matrices)
Lab 9	48	Recursion
Lab 10	53	Pointers
Lab 11	55	Algorithms (sorting, permutations, puzzles)
Lab 12	60	Structures (User-Defined data-types)

Almost Incorrect vs Almost Correct

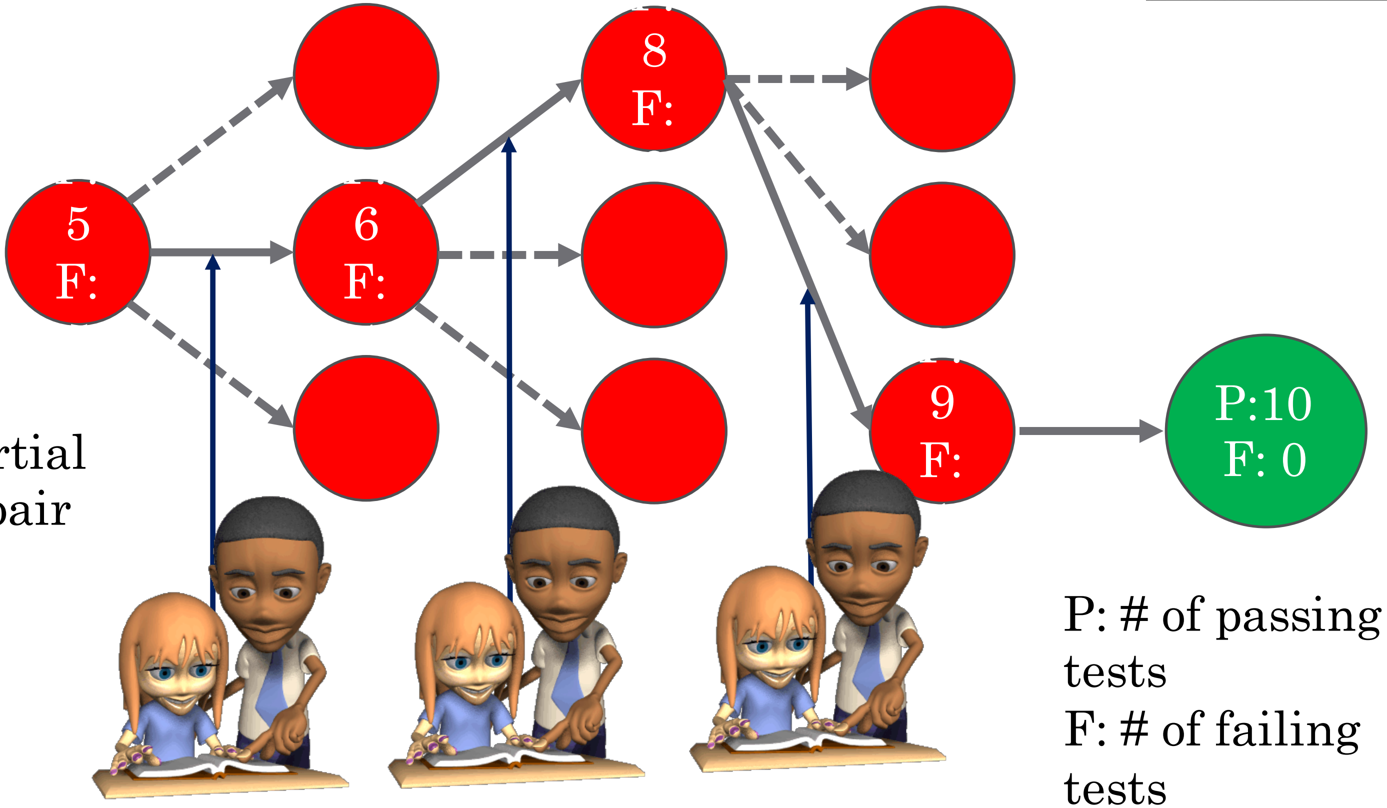


The fact that student programs are often significantly incorrect makes it difficult to fix those programs.

Tutoring ≠ Bug Fixing



Tailoring Repair Policy



partial
repair

P: # of passing
tests
F: # of failing
tests

Partial Repair: (all previously passing tests) + (at least one previously failing test)

Two-Step Repair

Test 1
Test 2
Test 3
Test 4
Test 5

```
+ if (true) {
+     S';
+ } else {
      S;
+ }
```

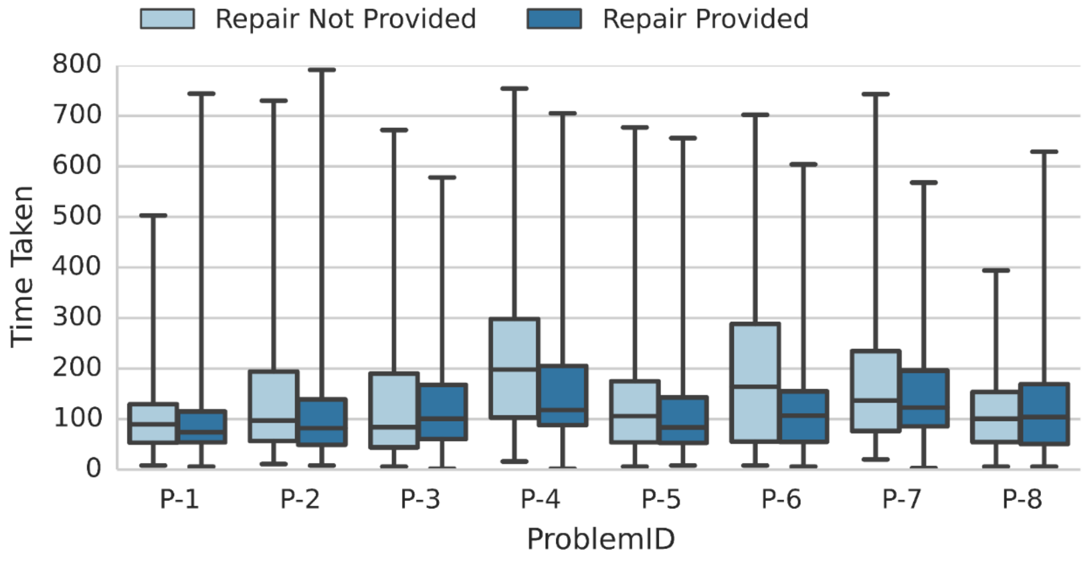
Test 1
Test 2
Test 3
Test 4
Test 5

```
+ if (E) {
+     S';
+ } else {
      S;
+ }
```

Test 1
Test 2
Test 3
Test 4
Test 5

User Study: Graders – Time Taken

- **43** buggy student submissions from dataset
  - Across 8 unique problems
- **37** TA graders volunteered for study
  - Each TA gets all 43 submissions to grade
  - With repair hints for half the submissions
- Task: Grade the buggy program
  - With marks on closeness to correct solution



Wrap up: Community Response

- Angelix (angelix.io) — program repair tool based on symbolic execution:
  - The first constraint-based repair systems that scales to large programs;
  - Repaired Heartbleed vulnerability in OpenSSL;46 stars on GitHub, 16 forks, 6 contributors;
  - Used by researchers from over 80 institutions; Used in intelligent tutoring system at IIT Kanpur.

- program-repair.org community website:
  - ~300 unique visitors per month;
  - ~100 researchers subscribed;
  - Contributors from ~10 institutions.

  - **The community is growing, please join and contribute!**

Relevant Research Results

Symbolic execution with second order existential constraints
Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, Abhik Roychoudhury
ACM Symposium on Foundations of Software Engineering (FSE) 2018.

Semantic Program Repair Using a Reference Implementation (PDF)
Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, Abhik Roychoudhury
ACM/IEEE 40th International Conference on Software Engineering (ICSE) 2018.

Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis (pdf)
Sergey Mechtaev, Jooyong Yi, Abhik Roychoudhury
ACM/IEEE International Conference on Software Engineering (ICSE) 2016.

DirectFix: Looking for Simple Program Repairs (PDF)
Sergey Mechtaev, Jooyong Yi, Abhik Roychoudhury
ACM/IEEE International Conference on Software Engineering (ICSE) 2015.

SemFix: Program Repair via Semantic Analysis (pdf)
Hoang D.T. Nguyen, Dawei Qi, Abhik Roychoudhury, Satish Chandra
ACM/IEEE International Conference on Software Engineering (ICSE) 2013.

http://www.comp.nus.edu.sg/~abhik/projects/Repair/index.html

ACKNOWLEDGEMENT: National Cyber Security Research program from NRF Singapore http://www.comp.nus.edu.sg/~tsunami/

Ack. to former students and grant

Marcel. Boehme, PhD. NUS 2014, Post-doc NUS -> Lecturer Monash

Van Thuan Pham, PhD. 2017

Sergey Mechtaev, PhD. 2018 -> Lecturer University College London

Shin Hwei Tan, PhD. 2018 -> Asst Prof, SUSTech, Shenzen. China

Jooyong Yi, past post-doc -> Asst Prof. Innopolis

ACKNOWLEDGEMENT: National Cyber Security Research program from NRF Singapore http://www.comp.nus.edu.sg/~tsunami/ and DSO National Labs