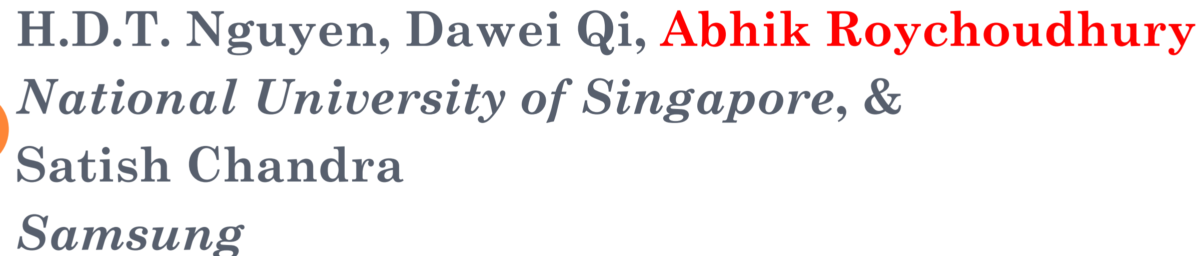
SemFix: Program Repair via Semantic Analysis



H.D.T. Nguyen, Dawei Qi, Abhik Roychoudhury
National University of Singapore, **&**
Satish Chandra
Samsung

Talk given at 30th CREST Workshop, London, Jan 2014.

What we have been discussing

Precise debugging is laborious.

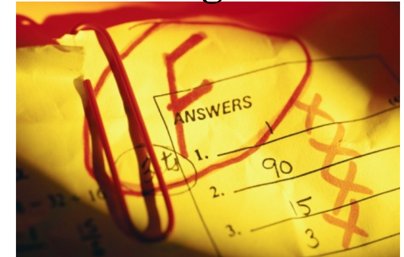
Specification based repair,
Genetic Programming,
…

Symbolic execution of test cases to extract specifications

THIS WORK …

Test–suite

Failing tests



Suspicions !! – statistical fault localization.

Infer intended meaning of suspicious statements
- Symbolic execution (SE)

Solve constraint from SE to create fixed statement
- Program synthesis

0. The problem

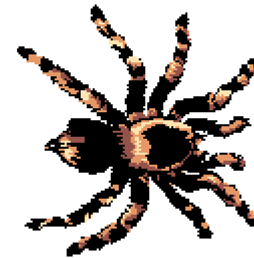
```
1   int is_upward( int inhibit, int up_sep, int down_sep){
2         int bias;
3         if (inhibit)
4             bias = down_sep; //   bias= up_sep + 100
5         else  bias = up_sep ;
6         if (bias > down_sep)
7              return 1;
8         else return 0;
9   }
```

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

1. Find a Suspect

```
1   int is_upward( int inhibit, int up_sep, int down_sep){
2         int bias;
3         if (inhibit)
4             bias = down_sep; //    bias= up_sep + 100
5         else  bias = up_sep ;
6         if (bias > down_sep)
7              return 1;
8         else return 0;
9   }
```

Line	Score	Rank
4	**0.75**	**1**
8	0.6	2
3	0.5	3
6	0.5	3
5	0	5
7	0	5



2 What it should have been

```
1   int is_upward( int inhibit, int up_sep, int down_sep){
2         int bias;
3         if (inhibit)
4             bias = down_sep; //   bias= up_sep + 100
5         else  bias = up_sep ;
6         if (bias > down_sep)
7              return 1;
8         else return 0;
9   }
```

inhibit	up_sep	down_sep	**Observed output**	**Expected Output**	**Result**
1	*11*	*110*	*0*	*1*	***fail***

Line 4

inhibit = 1, up_sep = 11, down_sep = 110
bias = X, path condition = true

Line 7

inhibit = 1, up_sep = 11, down_sep = 110
bias = X, path condition = X> 110

Line 8

inhibit = 1, up_sep = [illegible] down_sep = 110
bias = X, path c[illegible]iti[illegible] = X ≤ 110

2. What it should have been

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

```
1  int is_upward( int inhibit, int up_sep, int
   down_sep){
2       int bias;
3       if (inhibit)
4           bias = f(inhibit, up_sep, down_sep)
5       else  bias = up_sep ;
6       if (bias > down_sep)
7           return 1;
8       else return 0;
9  }
```

Symbolic Execution

f(1,11,110) > 110

CREST Workshop, Jan 2014

3. 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`

To Recapitulate

- Ranked Bug report
  - Hypothesize the error causes – suspect
- Symbolic execution
  - Specification of the suspicious statement
  - Input-output requirements from each test
  - Repair constraint
- Program synthesis
  - Decide operators which can appear in the fix
  - Generate a fix by solving repair constraint.

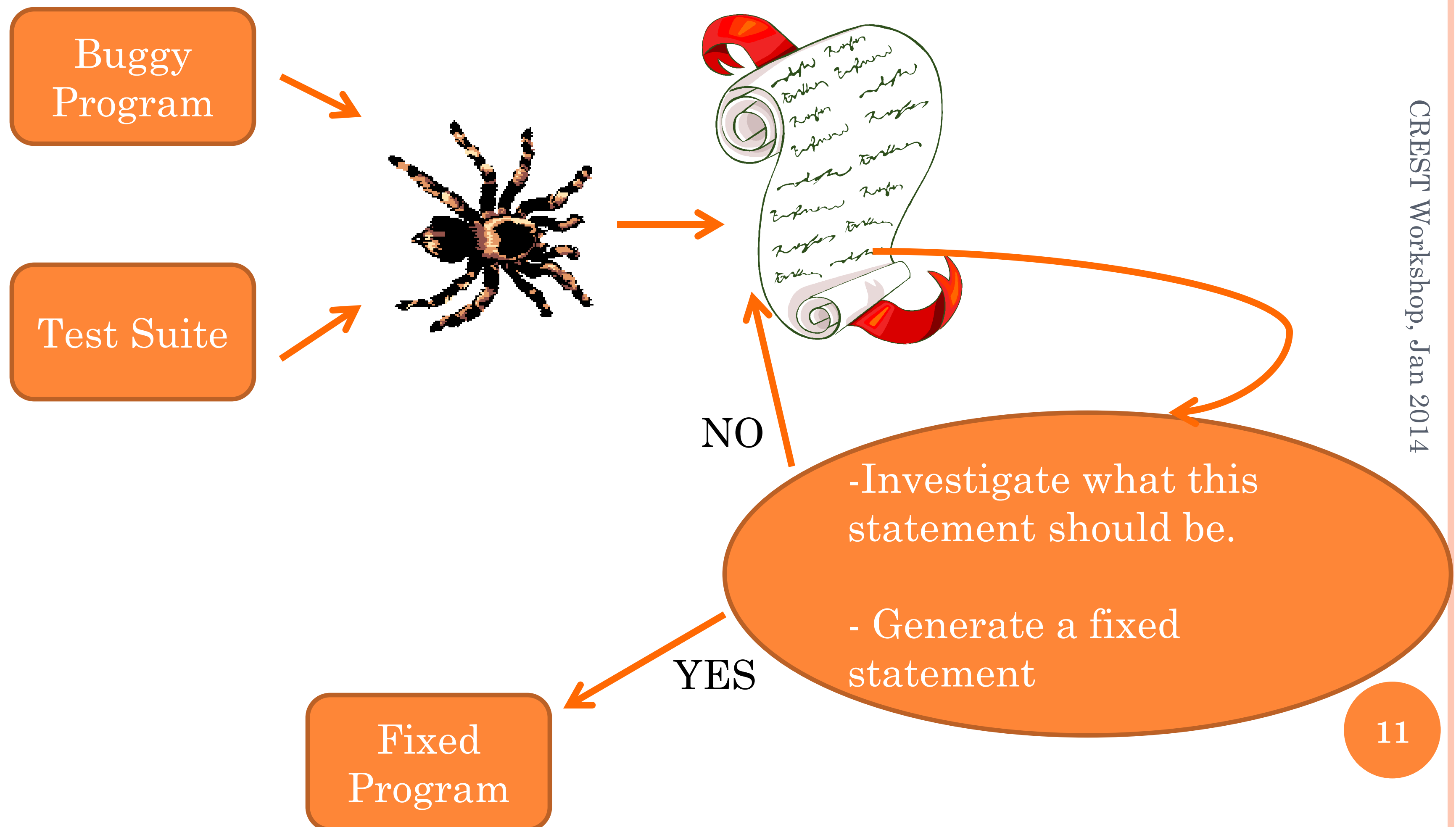
Producing Ranked Bug report

- We use the Tarantula toolkit.
- Given a test-suite T

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

  - fail(s) ≡ # of failing executions in which s occurs
  - pass(s) ≡ # of passing executions in which s occurs
  - allfail ≡ Total # of failing executions
  - allpass ≡ Total # of passing executions
    - allfail + allpass = |T|
- Can also use other metric like Ochiai.

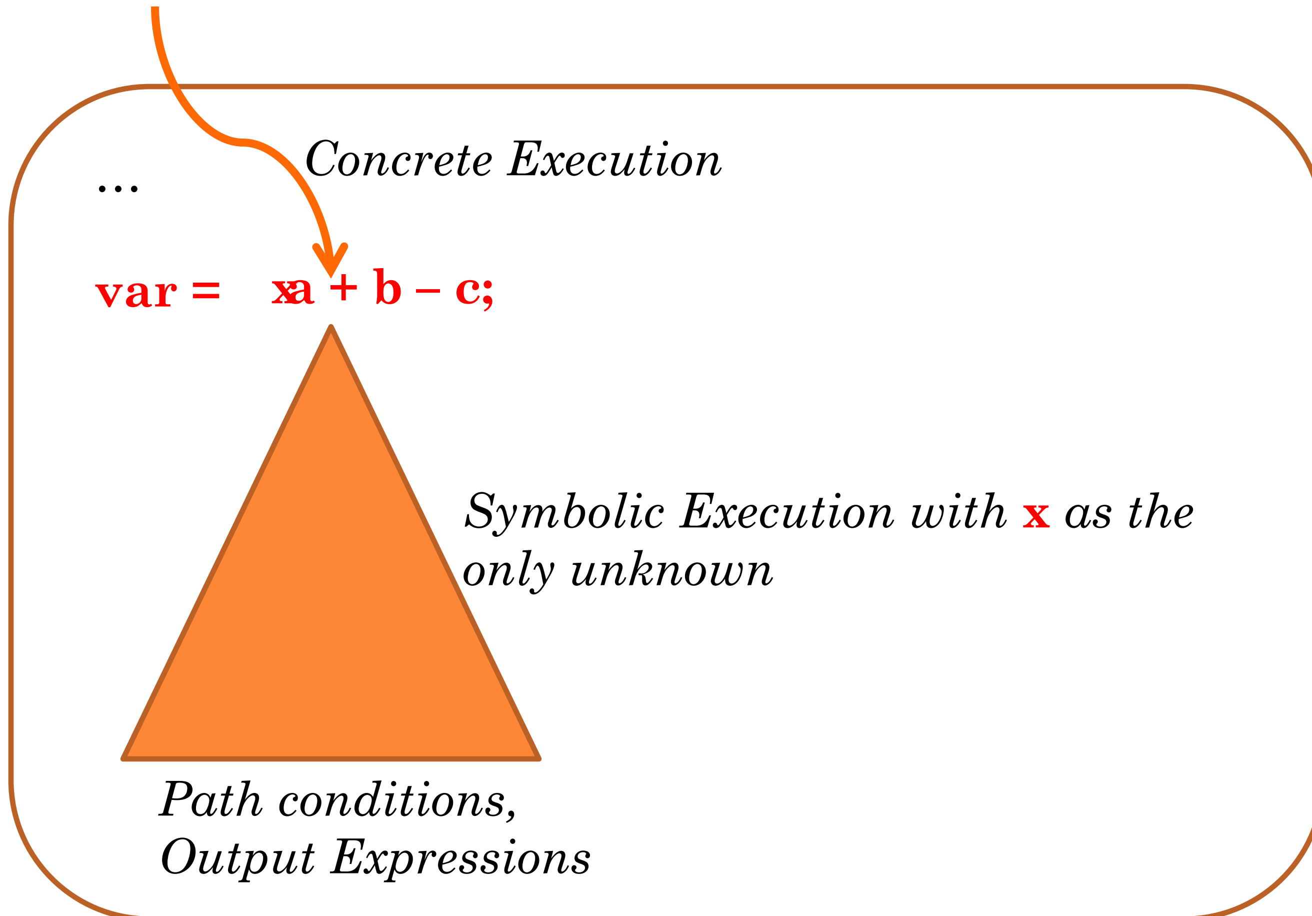
Usage of Ranked Bug report



To Recapitulate

- Ranked Bug report
  - Hypothesize the error causes – suspect
- Symbolic execution
  - Specification of the suspicious statement
  - Input-output requirements from each test
    - Repair constraint
- Program synthesis
  - Decide operators which can appear in the fix
  - Generate a fixed statement by solving repair constraint.

What it should have been

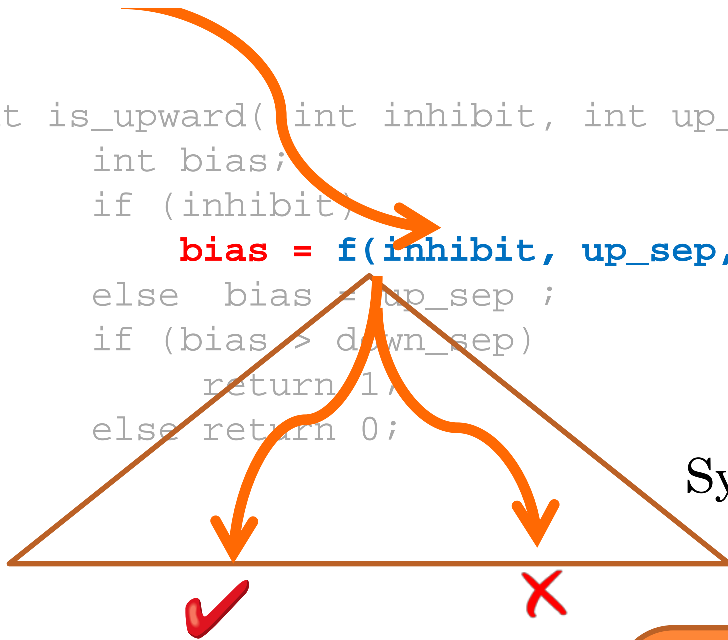


Buggy Program

Example

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

```
1  int is_upward( int inhibit, int up_sep, int down_sep){
2        int bias;
3        if (inhibit)
4            bias = f(inhibit, up_sep, down_sep)   // X
5        else   bias = up_sep ;
6        if (bias > down_sep)
7             return 1;
8        else return 0;
9  }
```



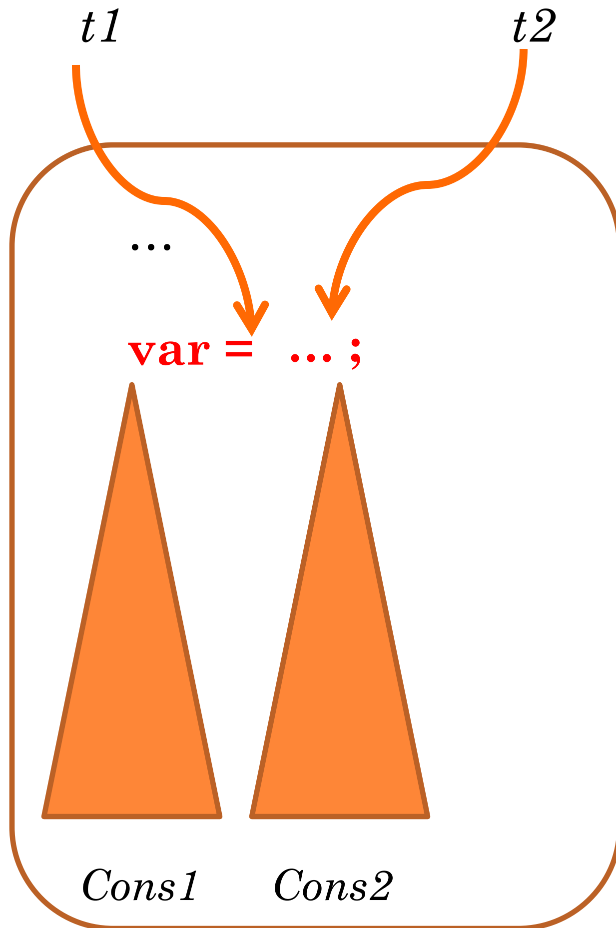
Symbolic Execution

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

$$\bigvee_{j \in Paths} (pc_j \wedge out_j == expected_out(t)) \wedge f(t) == X$$

Repair constraint

Overall Repair Constraint



Repair Constraint = *Cons1* $\wedge$ *Cons2*

$$\text{Repair constraint} = \bigwedge_{TS} \text{Cons}_i$$

1. ***TS*** = *failing tests;*
2. *Repair based on* ***TS*** *// guaranteed to pass TS*
3. *New = newly failed tests due to repair*
4. *If (New == ϕ) exit // Got your repair*
5. *else {* ***TS*** = ***TS*** $\cup$ *New;*
6. *Go to 2 }*

To Recapitulate

- Ranked Bug report
  - Hypothesize the error causes – suspect
- Symbolic execution
  - Specification of the suspicious statement
  - Input-output requirements from each test
  - Repair constraint
- Program synthesis
  - Decide operators which can appear in the fix
  - Generate a fix by solving repair constraint.

WHY PROGRAM 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
  - Where to place each component?

```
int tmp = down_sep -1;
return up_sep + tmp;
```

```
int tmp=down_sep + 1;
return tmp- inhibit;
```

  - What are the parameters?

```
int tmp = down_sep -1;
return tmp + inhibit ;
```

```
int tmp = down_sep -1;
return tmp + up_sep ;
```

Location Variables

- Define location variables for each component
- Constraint on location variables solved by SMT.
  - Well-formed e.g. defined before being used
  - Output constraint from each test (repair constraint)
  - Meaning of the components
  - Lines determine the value $L_x == L_y \Rightarrow x == y$
- Once locations are found, program is constructed.

Components = {+}
$L_{in} == 0$, $L_{out} == 1$, $L_{out+} == 1$, $L_{in1+} == 0$, $L_{in2+} == 0$

```
0  r0 = input;
1  r = r0 + r0;
2  return r;
```

EVALUATION

- Results from
  - SIR and GNU CoreUtils
- Tools
  - Ranked Bug report (Tarantula)
  - Symbolic execution (KLEE)
  - Program synthesis (Own tool + Z3)

Subjects Used

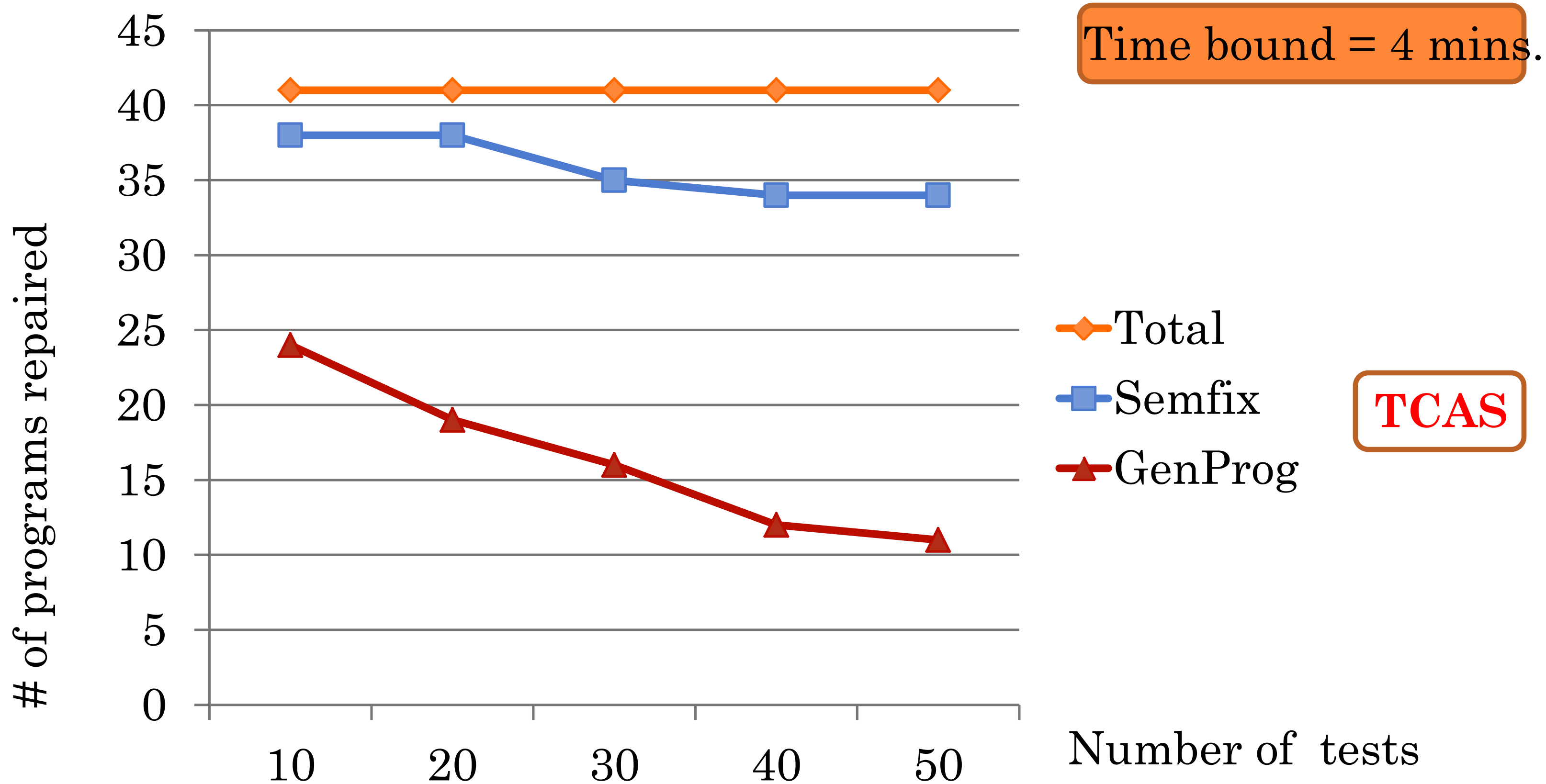
SIR programs

Subject	LoC	# Versions	Description
TCAS	135	41	Air Traffic Control
Schedule	304	9	Process scheduler
Schedule2	262	9	Process scheduler
Replace	518	29	Text processing
Grep	9366	2	Text search engine

GNU CoreUtils

Subject	LoC
mknod	183
mkdir	159
mkfifo	107
cp	2272

SUCCESS OF REPAIR (SIR)



Time bound = 4 mins.

TCAS

Overall 90 programs from SIR
SemFix repaired 48/90, GenProg repaired 16/90 for 50 tests.
GenProg running time is >3 times of SemFix

Type of Bugs (SIR)

	Total	SemFix	GenProg
Constant	14	10	3
Arithmetic	14	6	0
Comparison	16	12	5
Logic	10	10	3
Code Missing	27	5	3
Redundant Code	9	5	2
ALL	**90**	**48**	**16**

GNU CoreUtils

- 9 buggy programs where bug could be reproduced.
  - Taken from paper on KLEE, OSDI 2008.
- SemFix succeeded in 4/9 *[mkdir, cp, …]*
  - Average time = 3.8 mins.
  - Average time = 6 mins. [GenProg]
- All GenProg experiments using configuration from ICSE 2012 paper by Le Goues et al.
  - Pop size, # generations, …
  - Other configurations may lead to success for GP, but then we need a *systematic method* to determine the configurations.

Expression enumeration

- Enumerate all expressions over a given set of components (i.e. operators)
  - Enforce axioms of the operators
  - If candidate repair contains a constant, solve using SMT

- Program synthesis turns out to be faster.

Subject	TCAS	Schedule	Schedule 2	replace	grep
Ratio	6.9	2.8	2.5	1.36	2.2

Enumeration also timed out > 20 minutes. These are not even included.

REPAIRS THAT WERE NOT DONE

- Multiple line fix
  - Complex code to be inserted
  - Same wrong branch condition
    - `if (c ){ … } … if (c) { … }`
  - Updates to multiple variables
    - `x = e1; … ; y = e2; …`
- Floating point bugs
  - `n = (int) (count*ratio + 1.1);`
    - Can be overcome, limitation of KLEE/solvers
- Other problems, e.g. wrong function call
  - `current_job = (struct process *)0;` **`get_current();`**

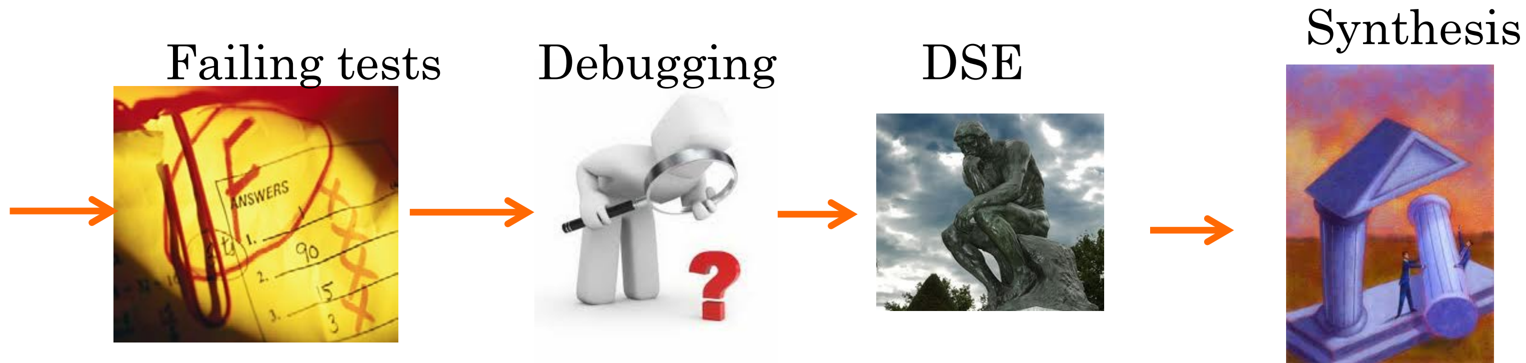
EXAMPLE FIXES

- `enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV); /*&& (Cur_Vertical_Sep > MAXALTDIFF);missing code*/`
  - Synthesizes missing code
- `tmp = Up_Separation;`
  - Synthesizes
  - `tmp =  ((OtherCapability < Alt_Layer_Value)?`
  - `Two_of_Three_Reports_Valid:`
  - `Cur_Vertical_Sep`
  - `);`

IN SUMMARY

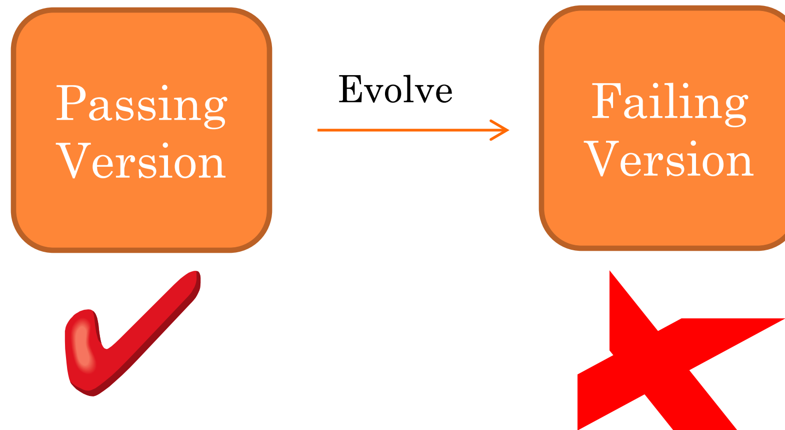
- Repair exploiting symbolic execution
  - Avoids enumeration over a space of expressions from a pre-fixed template language.
- Repair via constraint solving
  - Synthesize rather than lifting fixes from elsewhere.
- Repair without formal specifications
  - Pass given test cases by a constraint solver answering "*What it should have been*?"
- Single line repair – need to do more …
  - Try other background debugging tools / metrics.
  - Synthesize guards to relate different fragments to fix.

For Discussion - Ongoing



Minimized Mutations for Repair

For Discussion - Ongoing



Regression Repair

Research Questions
Can we use the changes as anchor to direct repair?
Is it possible to employ "mutations" at the change sites?

To investigate: it may sometimes be easier to make multiple simple repairs, rather than one-line complex repair, a-la SEMFIX.