DirectFix: Looking for Simple Program Repairs

Sergey Mechtaev   Jooyong Yi   Abhik Roychoudhury

School of Computing
National University of Singapore

May 21, 2015

Repair problem

How to formulate the **test-driven program repair problem**?

Variant 1:

> *Given a test suite T and a buggy program P, find a program P' that passes T.*

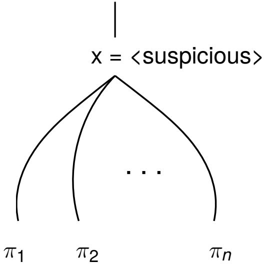
(implied by most existing repair approaches)

SemFix (ICSE'13)

1. Localizes suspicious statement using statistical fault localization.

2. Infers specification for test case (i, o):

$$\bigvee_{j} (\pi_j \wedge \text{input} = i \wedge \text{output} = o)$$



3. Synthesizes desired expression using constraint-based program syntheisis.

Problem:

- only single-line fixes.

Genprog (ICSE'09)

Syntactical search-based repair approach.

> *Local search (genetic programming) swapping, inserting and deleting existing program statements guided by the number of passing test cases.*

Problem:

- complicated unmaitainable patches.

Many solutions

If test suite $T = \{(\text{input}_1, \text{output}_1,), (\text{input}_2, \text{output}_2,), ...\}$, then

```
if (input_1) {
  return output_1;
} else if (input_2) {
  return output_2;
} else ...
```

is a valid solution.

Conclusion:

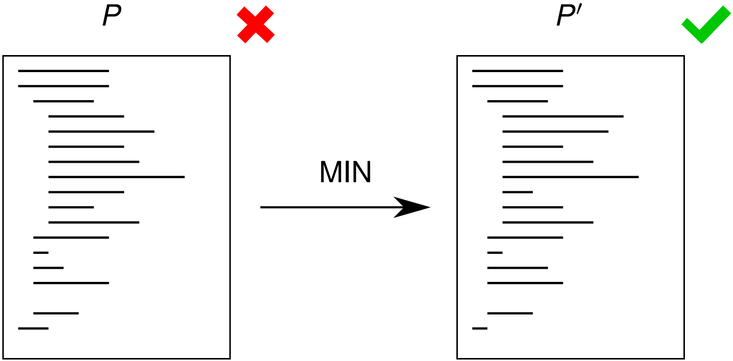
> *There are many ways to fix the bug. Most of them are unsatisfactory.*

High quality automatic patches:

- easily understandable by developers;
- don't break functionality that isn't covered by test suite.

Minimality

*Look for the **minimal change** of the source code that fixes the bug.*



Repair problem revised

How to formulate the **test-driven program repair problem**?

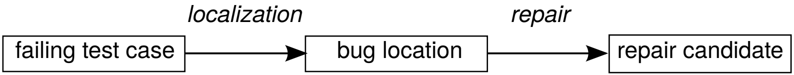
Variant 2:

Given a test suite T and a buggy program P, find a program P' that

- *passes T;*
- *syntactically closest to P.*

Avoiding bug localization

Traditional approach:



failing test case

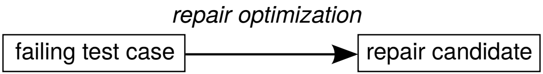
localization

bug location

repair

repair candidate

Our approach:



failing test case

repair optimization

repair candidate

Correctness

Bugs cause non-fulfillment of given **requirement.**

Example:

$$\underbrace{(x^2 + 3x + 1)}_{\text{implementation}} \qquad \underbrace{(x+1)^2}_{\text{intention}}$$

We expect

$$\forall x.\, x^2 + 3x + 1 = (x+1)^2$$

which is false.

Repair through satisfiability

Buggy program:

$$x^2 + 3x + 1$$

Parametrize implementation:

$$\forall x.\, x^2 + ax + b = (x+1)^2$$

SMT solver:

$$a = 2,\ b = 1 \qquad ✓$$

could be several solutions, any of them exactly corresponds to our intentions.

- usually, we don't have formal specification.

Test-driven repair

Buggy program:

$$x^2 + 3x + 1$$

For the test case $(1, 4)$, we expect

$$x^2 + ax + b = r \wedge \underbrace{x = 1 \wedge r = 4}_{\text{test case}}$$

SMT solver:

$$a = 2,\ b = 1 \qquad ✓$$

or...

$$a = 1,\ b = 2 \qquad ✗$$

which corresponds to different function.

- breaks unspecified functionality.

Minimal change using MaxSAT

Buggy program:

$$x^2 + 3x + 1$$

Repair condition (RC)

$$\underbrace{x^2 + ax + b = r \land x = 1 \land r = 4}_{\text{hard constraints}} \land \underbrace{a = 3 \land b = 1}_{\text{soft constraints}}$$

binds syntax and semantics.

MaxSMT solver:

$$a = 2,\ b = 1 \quad ✓$$

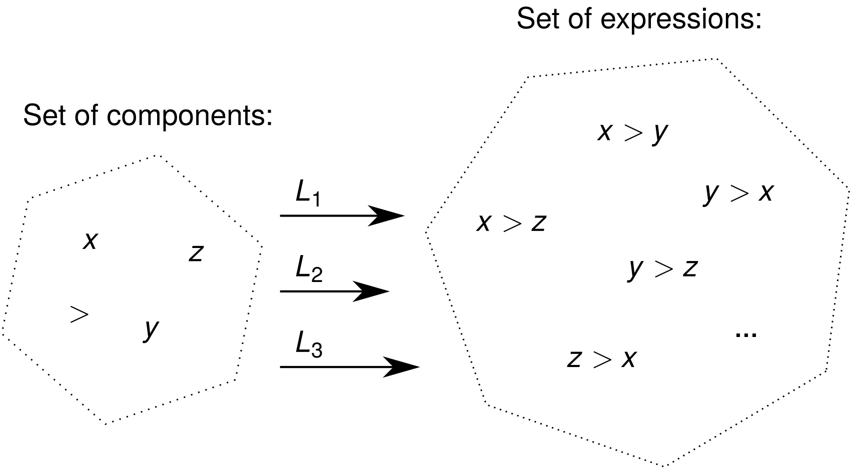
could be many solutions, not any of them exactly corresponds to our intentions.

However,

- breaks less unspecified functionality.

Component-based synthesis (ICSE'10)

synthesis($+, -, \times, ...,$ location variables L)



Synthesis workflow

synthesis$(x, y, >, L)$ → SMT solver

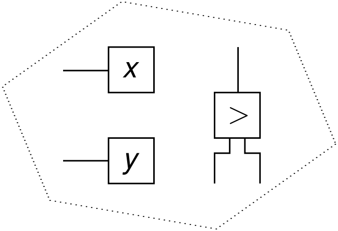
$$L(>^{out}) = 2$$

$$L(>^{in}_1) = 0$$

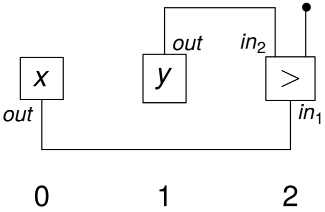
$$L(>^{in}_2) = 1$$

$$L(x^{out}) = 0$$

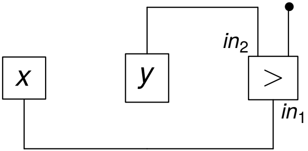
$$L(y^{out}) = 1$$



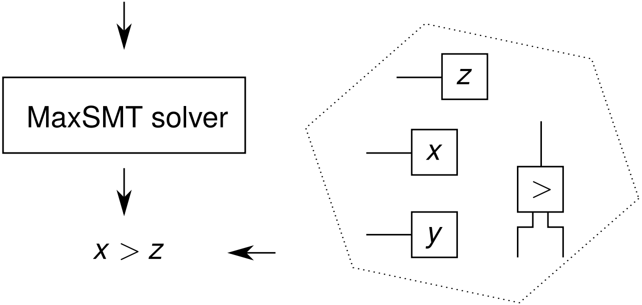
+



Repair workflow



$$\underbrace{\text{synthesis}(x, y, z, >, L)}_{\text{hard constraints}} \wedge \underbrace{L(>_1^{in}) = L(x) \wedge L(>_2^{in}) = L(y)}_{\text{soft constraints}}$$



MaxSMT solver

$x > z$

Repair condition

For a given expression consisting of components $+$, $-$, $\times$, **...**, **repair condition** is

$$\underbrace{\text{synthesis}(+,-,\times,...,L)}_{\text{hard constraints}} \wedge \underbrace{\text{connections}}_{\text{soft constraints}}$$

Whole program

Program formula F:

```
if (x > y) {
  y = y + 1;
} else {
  y = y - 1;
}
return y + 2;
```

$$\begin{aligned} F = (&\text{if}(x_1 > y_1) \\ &\text{then } (y_2 = y_1 + 1) \\ &\text{else } (y_2 = y_1 - 1)) \\ &\wedge (\mathit{result} = y_2 + 2) \end{aligned}$$

Program repair condition:

$$F[e_1 \leftarrow v_1, ..., e_k \leftarrow v_k] \wedge v_1 = RC(e_1) \wedge ... \wedge v_k = RC(e_k)$$

where e_1, ..., e_k are program expressions.

Performance

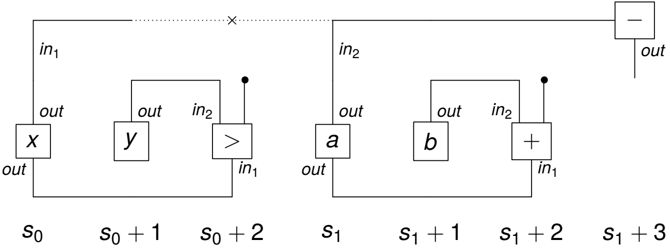
Synthesis vs Repair

It is significantly faster to repair an existing program than to synthesize a new one if the required change is small.

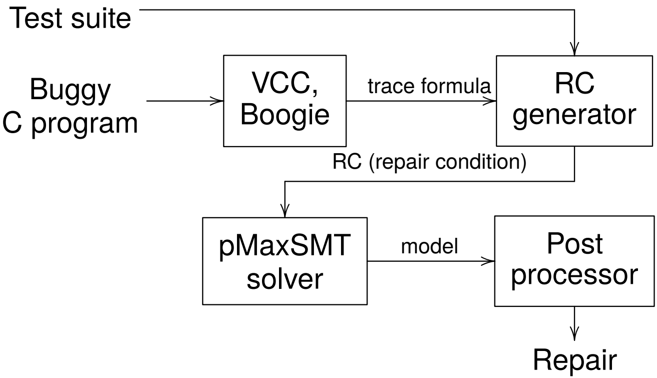
Optimization

Problem: too many components to consider for each program expression.

Solution: share component between several expressions.



Implementation



Subject programs

Subject	LOC	#Versions	Description
Tcas	135	41	Air traffic control program
Replace	518	30	Text processor
Schedule	304	9	Process scheduler
Schedule2	262	9	Process scheduler
Coreutils (mkfifo, mkdir, mknod, cp, pr, ptx, tac, md5sum, paste)	107 – 2909	9	Collection of OS utilities

Evaluation results

Subject	Total	DirectFix				SemFix (ICSE'13)			
		E	S	D	R	E	S	D	R
Tcas	30	16	29	2.26	12	3	11	4.1	17
Replace	5	5	5	2.8	0	3	4	10.2	2
Schedule	4	2	4	2.5	1	1	4	8.5	3
Schedule2	2	1	2	2	1	1	2	5	2
Coreutils	4	0	3	2	-	0	0	4	-
Overall	44	53%	95%	2.31	31%	17%	46%	6.36	54%

Legend: **E**quivalent, **S**ame location, **D**iff, **R**egression

Example (tcas)

DirectFix multi-line patch:

```
bool Own_Below_Threat() {
  /* BEFORE: <= */
  return (Own_Tracked_Alt < Other_Tracked_Alt);
}

bool Own_Above_Threat() {
  /* BEFORE: <= */
  return (Other_Tracked_Alt < Own_Tracked_Alt);
}
```

Example (replace)

DirectFix patch:

```
bool omatch(char *lin, int *i, char *pat, int j) {
  ...
  /* BEFORE: j */
  if ((lin[*i] != NEWLINE) && (!locate(lin[*i], pat, j + 1)))
  ...
}
```

SemFix patch:

```
while (i > offset)
  /* BEFORE: c == pat[i] */
  if (i < 6) { flag = true; i = offset; }
  else i = i - 1;
```

Summary

- Semantical program repair approach.
- Produces multi-line patches.
- Produces high-quality patches:

  minimizes syntactical change.
- Effective:

  avoids imprecise bug localization.