

# Automatic Program Repair (part 1)

# We've spent a lot of time on *finding* bugs.

- What about *fixing* them?
- Problem: Given a program and an indication of a bug, find a patch for that program to fix that bug.
  - Both static and dynamic techniques have been used to “indicate” bugs.
  - The bulk of repair research is *dynamic*, or uses tests.
  - (We'll talk about static briefly, and again later.)



# Bug fixing: the 30000-foot view

1. Localize the bug
  - And perform additional analysis
2. Create/combine fix possibilities into 1+ possible patches.
3. Validate candidate patches.



Fault  
localization



Tests.







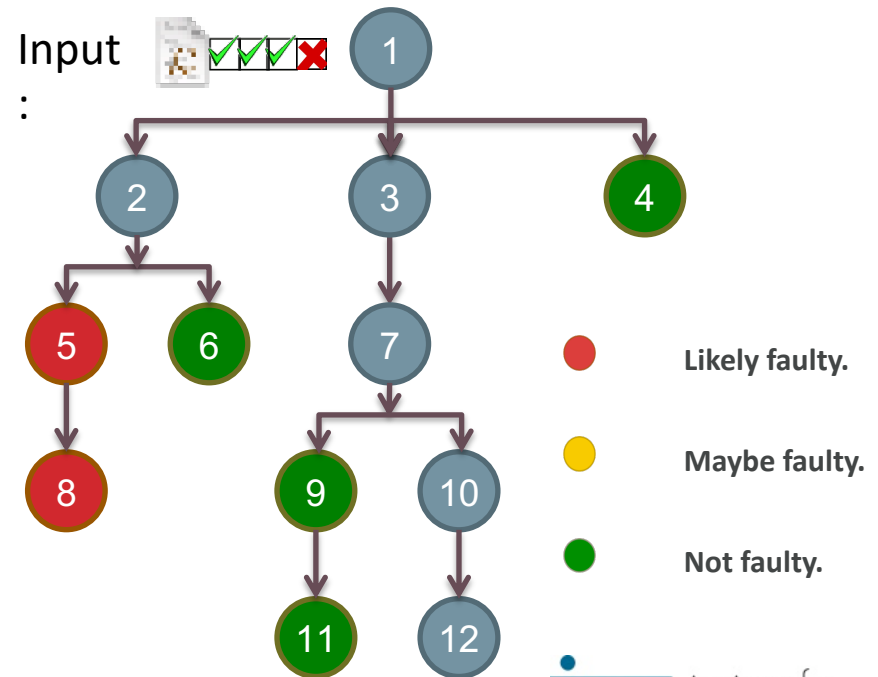
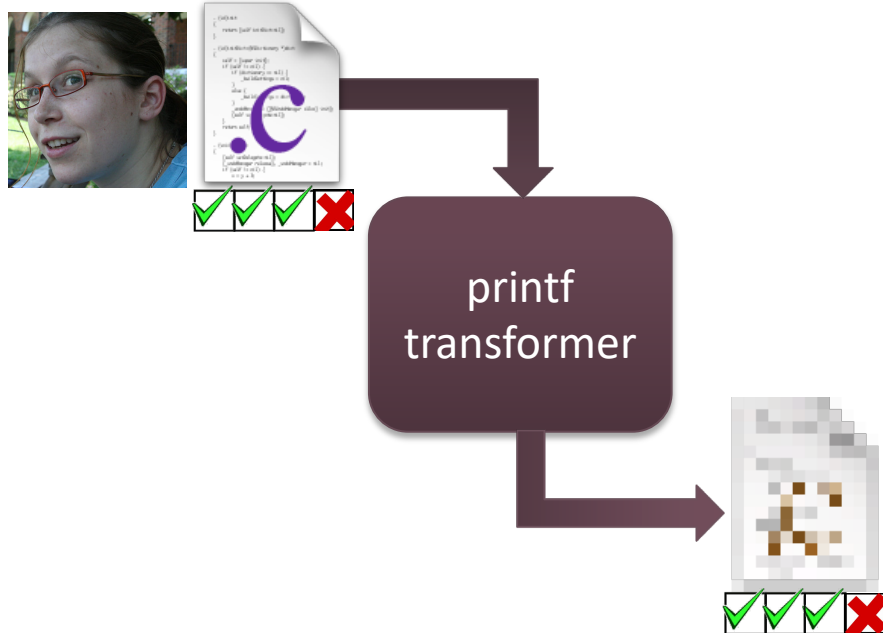
```
... (void) x;  
C  
return [self autorelease];  
}  
... (void) initWithDictionary:(NSDictionary *)dict  
C  
self = [super init];  
if (self != nil) {  
    if ([dictionary isKindOfClass:[NSDictionary class]])  
        [self initWithDictionary:dictionary];  
    else {  
        NSLog(@"initWithDictionary: unrecognized object");  
        return nil;  
    }  
}  
return self;  
}  
... (void) initWithDictionary:(NSDictionary *)dict  
C  
if (self != nil) {  
    if ([dictionary isKindOfClass:[NSDictionary class]])  
        [self initWithDictionary:dictionary];  
    else {  
        NSLog(@"initWithDictionary: unrecognized object");  
        return nil;  
    }  
}
```



printf  
transformer

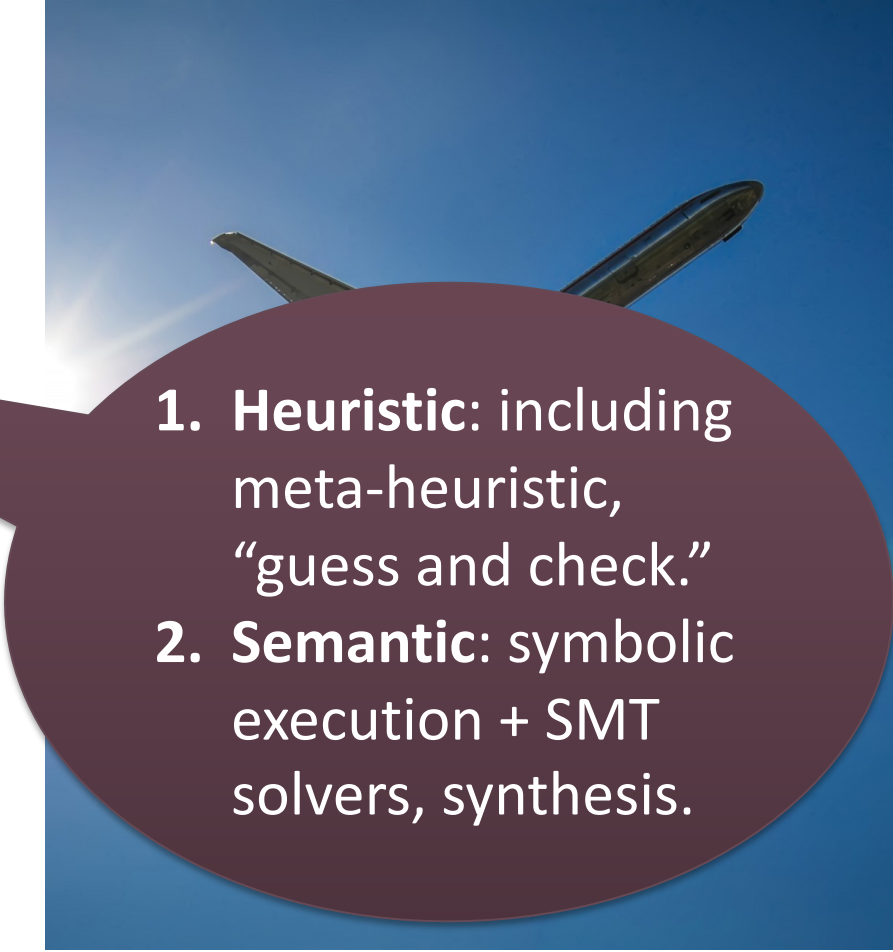


# Spectrum-based fault localization automatically ranks potentially buggy program pieces based on test case behavior.



# Bug fixing: the 30000-foot view

1. Localize the bug.
  - And perform additional analysis
2. Create/combine fix possibilities into 1 possible patches.
3. Validate candidate patch.

- 
1. **Heuristic:** including meta-heuristic, “guess and check.”
  2. **Semantic:** symbolic execution + SMT solvers, synthesis.

# GenProg: meta-heuristic

1. Localize the bug.
  - And perform additional analysis
2. Create/combine fix possibilities into a possible patch.
3. Validate candidate patch.

Localize to C statements.

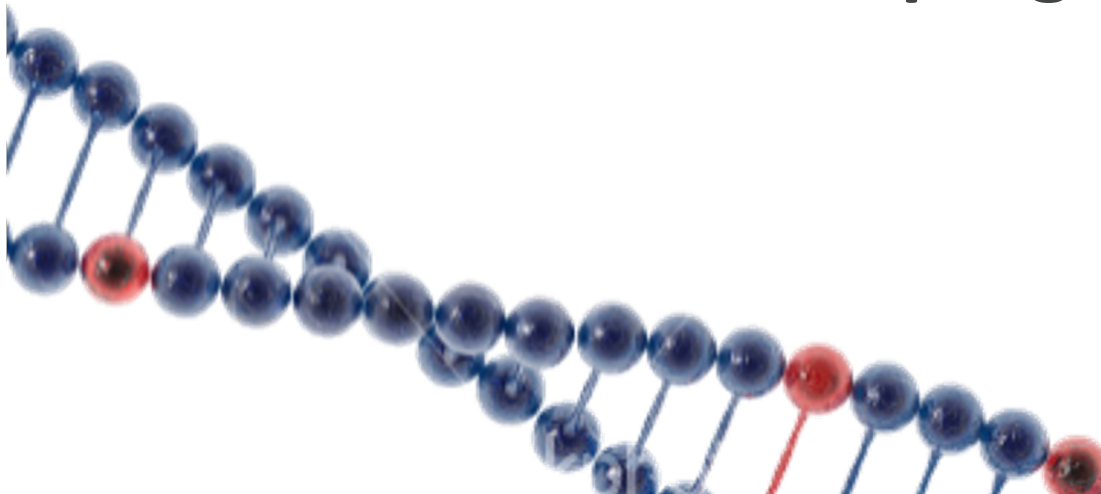
Use genetic programming to search for statement-level patches, reusing code from existing program.

# GenProg: automatic program repair with evolutionary computation.

Biased, random search for a AST-level edits to a program that fixes a given bug without breaking any previously-passing tests.



**Genetic programming:** the application of evolutionary or **genetic algorithms** to program source code.







# INPUT

The input stage shows a document icon containing C code. Below the code is a horizontal bar with four checkboxes. The first three are checked (green), and the fourth is unchecked (grey with an 'X').

# EVALUATE FITNESS

The evaluate fitness stage shows a document icon with C code placed on a platform scale. The scale has four indicators above it: three green checkmarks and one red 'X'. Below the platform is a circular dial with a red needle pointing to the right.

DISCARD



ACCEPT

The mutate stage shows a collection of document icons representing a population of code files. Some are white with a large 'C', some are grey, and one is a pixelated version of the 'C'. A central cluster of small grey dots represents a mutation process.

# MUTATE

The accepted stage shows a document icon with C code. Below it is a horizontal bar with four checked checkboxes (green).



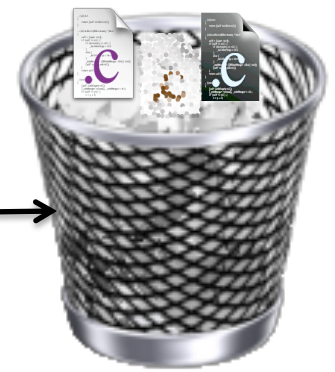
# INPUT

The input stage shows a document with C code. Below the document is a fitness bar consisting of four boxes: the first three contain green checkmarks, and the fourth contains a red X.

# EVALUATE FITNESS

The evaluate fitness stage shows a document with C code placed on a scale. Below the scale is a clock face with a red hand pointing to approximately 10:10.

DISCARD



ACCEPT

The mutate stage shows a collection of documents with various colored C logos (purple, blue, green, brown, red, black). A central cluster of small grey dots is also present.

# MUTATE

The output stage shows a document with C code and a large red 'C' logo. Below the document is a fitness bar consisting of four boxes, each containing a green checkmark.

# INPUT

The input stage features a document icon containing C code. Below the code is a horizontal bar with four checkboxes. The first three checkboxes are checked, and the fourth is unchecked with an 'X' over it.

# EVALUATE FITNESS

The evaluate fitness stage shows a document icon with C code placed on a platform scale. A clock face is positioned below the scale, indicating the time taken for evaluation.

DISCARD



ACCEPT

The mutate stage displays a collection of code files with various colored dots (purple, green, red, brown) scattered around them, representing the generation of new offspring from the current population.

# MUTATE

The accepted stage shows a single code file with a large 'C' logo. Below it is a horizontal bar with four checked checkboxes, indicating that this individual has met the fitness criteria.

# An individual is a candidate patch/set of changes to the input program.

- A patch is a series of *statement-level* edits:
  - delete X
  - replace X with Y
  - insert Y after X.
- Replace/insert: pick Y from somewhere else in the program.
- To mutate an individual, add new random edits to a given (possibly empty) patch.
  - (Where? Right: fault localization!)

>

```
1 void gcd(int a, int b) {
2     if (a == 0) {
3         printf("%d", b);
4     }
5     while (b > 0) {
6         if (a > b)
7             a = a - b;
8         else
9             b = b - a;
10    }
11    printf("%d", a);
12    return;
13 }
```

> gcd(4, 2)

> 2

>

> gcd(1071, 1029)

> 21


>

> gcd(0, 55)

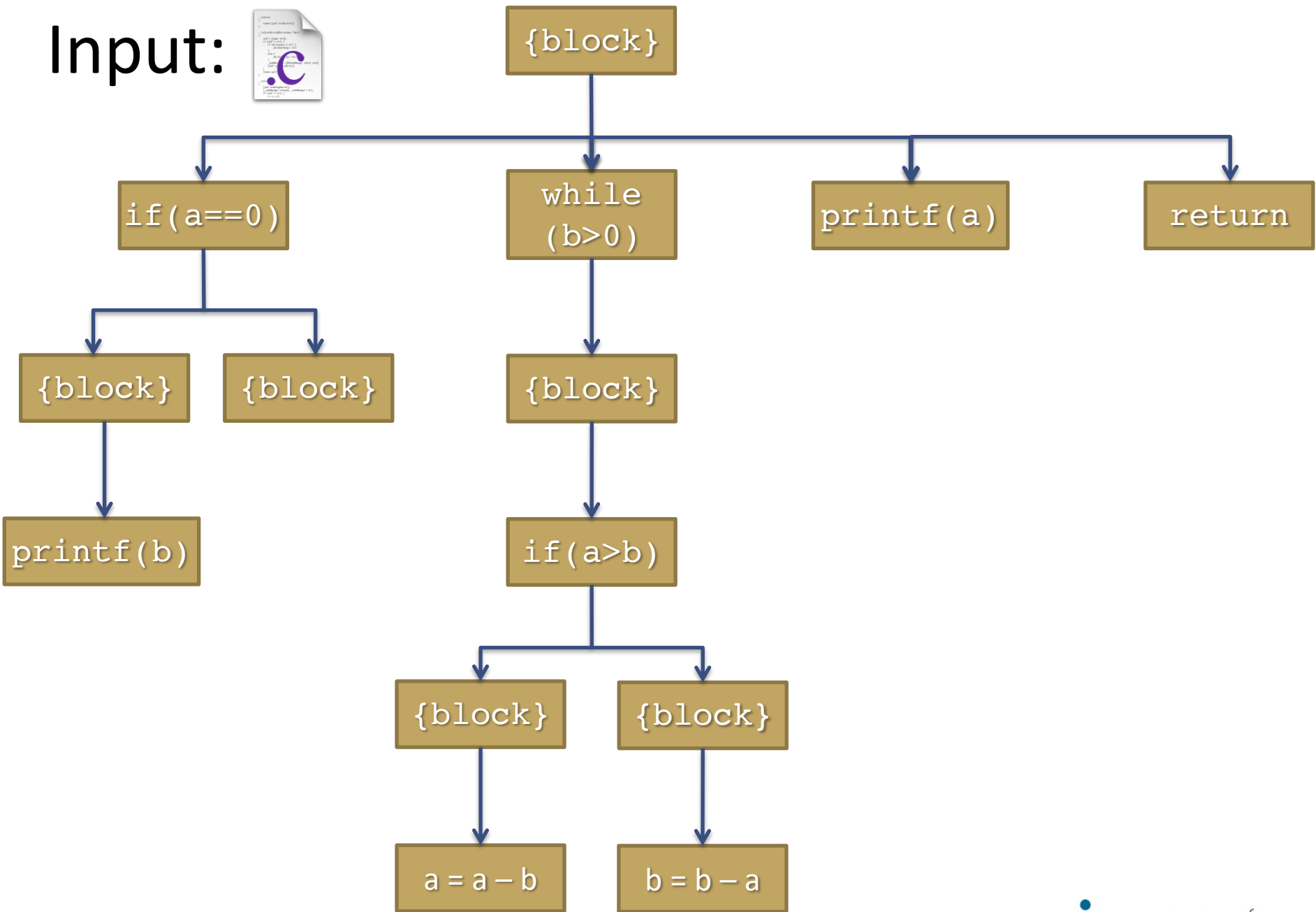
> 55

(looping forever)

```
1 void gcd(int a, int b) {
2     if (a == 0) {
3         printf("%d", b);
4     }
5     while (b > 0) {
6         if (a > b)
7             a = a - b;
8         else
9             b = b - a;
10    }
11    printf("%d", a);
12    return;
13 }
```



# Input:



Input: 

{block}

if (a==0)

while (b>0)

printf(a)

return


{block}

{block}

{block}

Legend:

 High change probability.

 Low change probability.

 Not changed.

printf(b)

if (a>b)

{block}

{block}

a = a - b

b = b - a

Input:

{block}

if (a==0)

while (b>0)

printf(a)

return

{block}

{block}

{block}

if (a>b)

printf(b)

{block}

{block}

a = a - b

b = b - a

An edit is:

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



Input:

{block}

if (a==0)

while (b>0)

printf(a)

return

{block}

{block}

{block}

printf(b)

if (a>b)

{block}

{block}

a = a - b

b = b - a

An edit is:

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

Input:

{block}

if (a==0)

while (b>0)

printf(a)

return

{block}

{block}

{block}

printf(b)

if (a>b)

{block}

{block}

a = a - b

b = b - a

An edit is:

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

Input:

{block}

if (a==0)

while (b>0)

printf(a)

return

{block}

{block}

{block}

An edit is:

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

printf(b)

return

if (a>b)

{block}

{block}

a = a - b

b = b - a

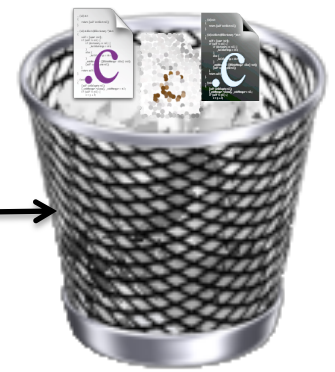
# INPUT

The input stage shows a document icon containing C code. A large purple 'C' logo is overlaid on the code. Below the document is a row of four checkboxes: the first three are green and checked, and the fourth is red and unchecked.

# EVALUATE FITNESS

The evaluate fitness stage shows a document icon with C code and a blue 'C' logo. The document is placed on a scale with a clock face below it, indicating the process of measuring fitness.

DISCARD

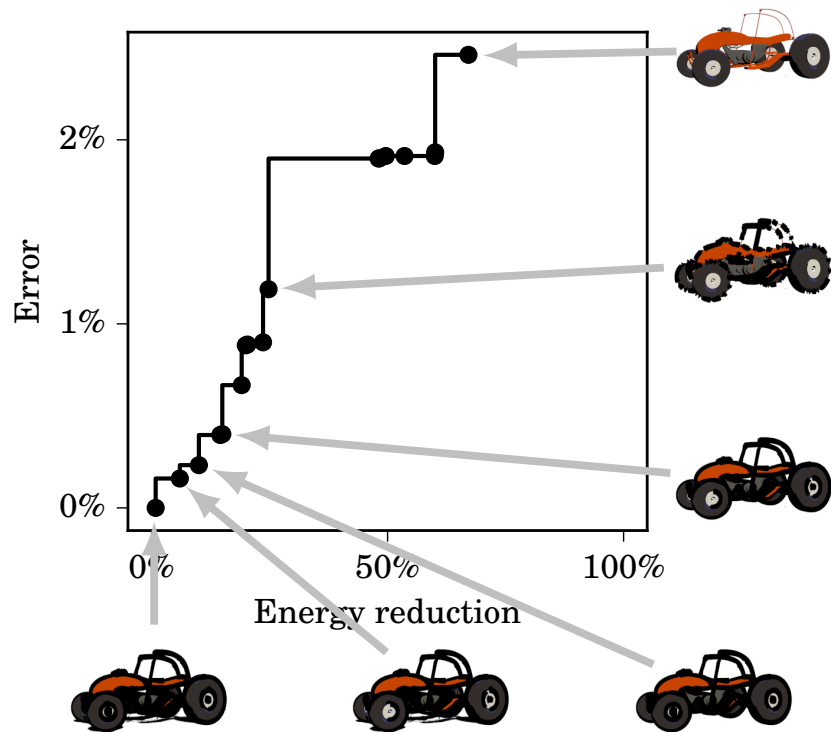


ACCEPT

The mutate stage shows a collection of code files with various colored 'C' logos (purple, blue, green, brown, red, black). A central cluster of small grey dots represents the genetic material being mutated.

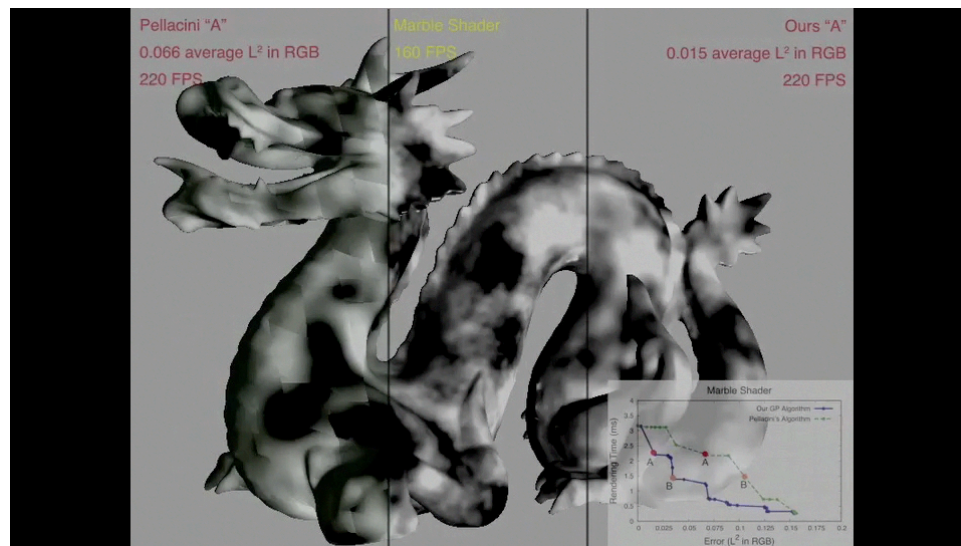
# MUTATE

The output stage shows a document icon with C code and a red 'C' logo. Below the document is a row of four green checked checkboxes, indicating that the selected individual meets all criteria.



Dorn et al. "Automatically Exploring Tradeoffs Between Software Output Fidelity and Energy Costs." IEEE Transactions on Software Engineering, vol. PP, no. 99, pp. 1–1, Nov. 2017.

Sitthi-amorn et al. "Genetic Programming for Shader Simplification." ACM Transactions on Graphics (Proc. SIGGRAPH Asia) 30(6): 152 (2011)



# Semantics-based repair

1. Localize the bug.
  - And perform additional analysis
2. Create/combine fix possibilities into 1+ possible patches.
3. Validate candidate patch.

Same idea, but localizing to *expressions*.

RHS of assignments, conditionals.

```
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 }
```



(Tremendous gratitude to Abhik Roychoudhury for sharing slides with me.)

```

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
<b>1</b>	<b>11</b>	<b>110</b>	<b>0</b>	<b>1</b>	<b>fail</b>
0	100	50	1	1	pass
<b>1</b>	<b>-20</b>	<b>60</b>	<b>0</b>	<b>1</b>	<b>fail</b>
0	0	10	0	0	pass



# What about Angelix?

1. Localize the bug.
  - And perform analysis
2. Create/combine fix possibilities into 1+ possible patches.
3. Validate candidate patches.

*Concolic execution to find expression values that would make the test pass.*

*Program synthesis to construct replacement code that produces those values.*

# An expression's *angelic value* is the value that would make a given test case pass.

- This value is set “arbitrarily”, by which we mean symbolically.
- You can *solve* for this value if you have:
  - the test case’s expected input/output.
  - the path condition controlling its execution.
- Concolic execution (remember me?):
  - Start executing the test concretely, and then switch to symbolic execution when the angelic value starts to matter.

```

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
<b>1</b>	<b>11</b>	<b>110</b>	<b>0</b>	<b>1</b>	<b>fail</b>
0	100	50	1	1	pass
<b>1</b>	<b>-20</b>	<b>60</b>	<b>0</b>	<b>1</b>	<b>fail</b>
0	0	10	0	0	pass

```

1  int is_upward( int inhibit, int up_sep, int down_sep){
2      int bias;
3      if (inhibit)
4  →      bias =  $\alpha$ ; // 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

inhibit = 1, up\_sep = 11, down\_sep = 110  
 bias =  $\alpha$ , PC = true

Line 4

Line 7

inhibit = 1, up\_sep = 11, down\_sep = 110  
 bias =  $\alpha$ , PC =  $\alpha > 110$

Line 8

inhibit = 1, up\_sep = 11, down\_sep = 110  
 bias =  $\alpha$ , PC =  $\alpha > 110$

# Collect all of the constraints!

- Accumulated constraints over all test cases:

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

- Use **oracle guided component-based program synthesis** to construct satisfying  $f$ :

– *How does this work again?*

- Generated fix

$$- f(\text{inhibit}, \text{up\_sep}, \text{down\_sep}) = \text{up\_sep} \\ + 100$$



# Trick to multi-expression repair: “forests”

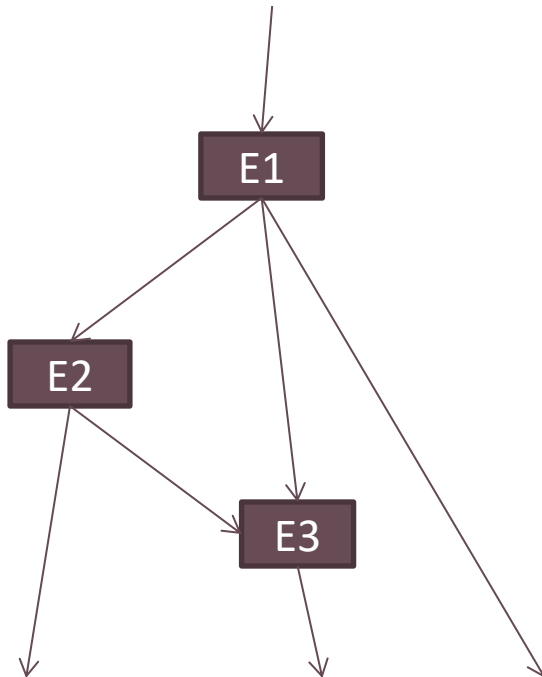




# Angelic Forest

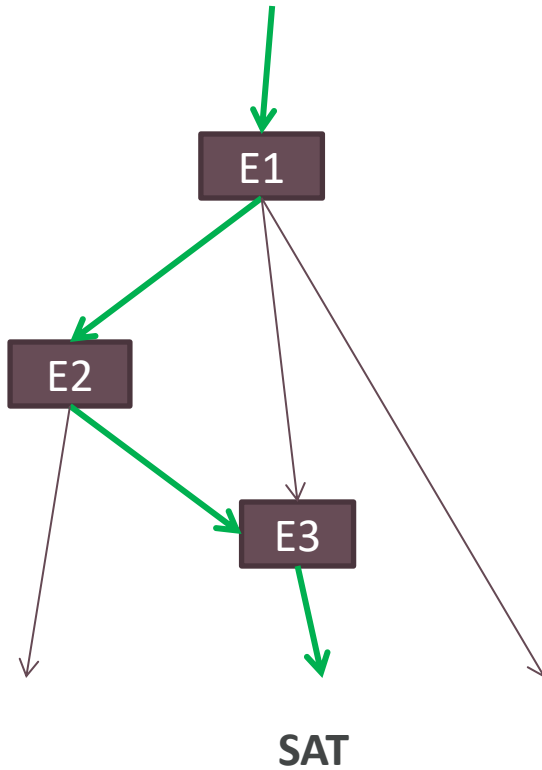
Program

Angelic Paths



# Angelic Forest

Program



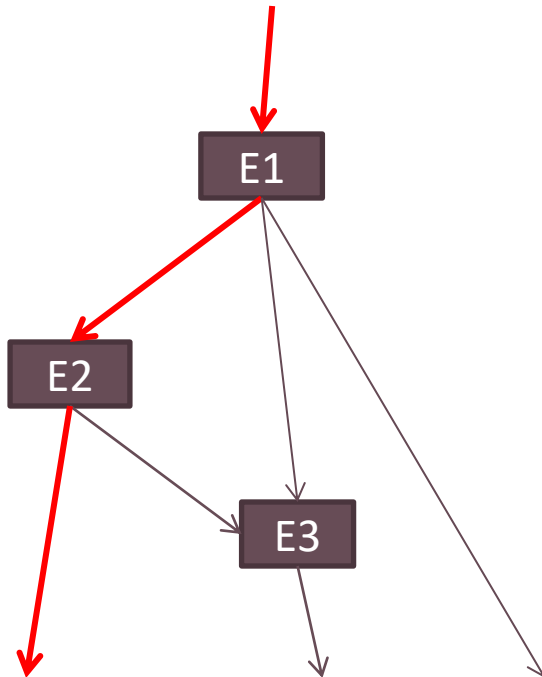
Angelic Paths





# Angelic Forest

Program



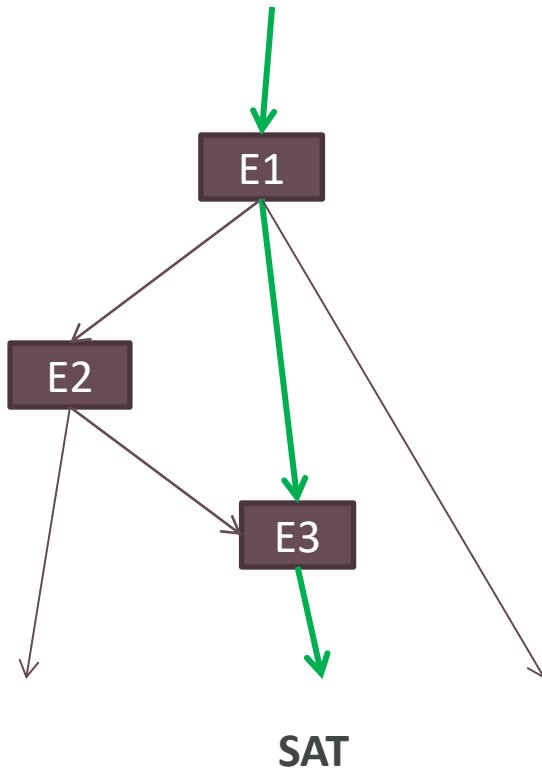
UNSAT

Angelic Paths

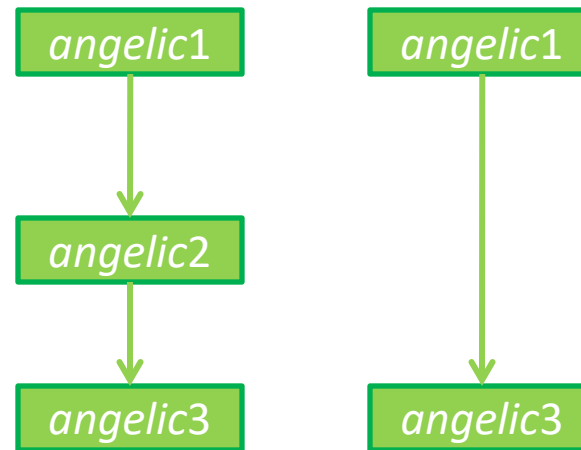


# Angelic Forest

Program

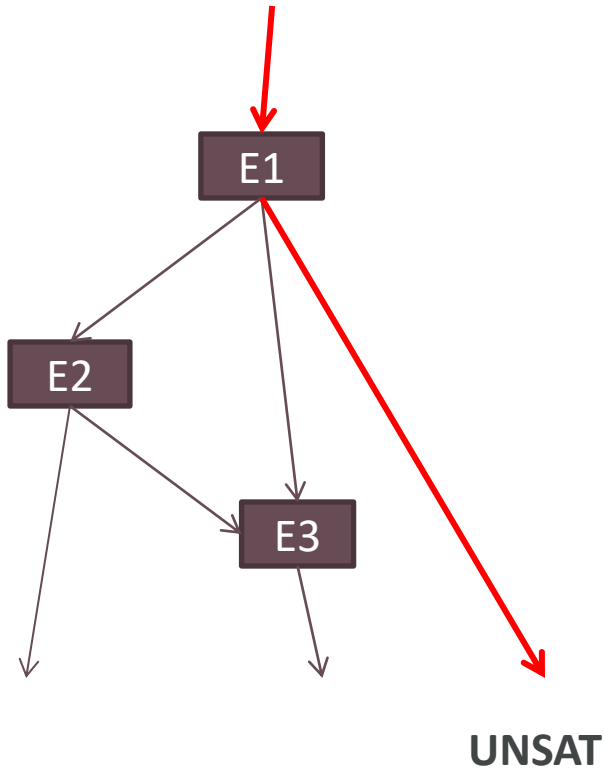


Angelic Paths

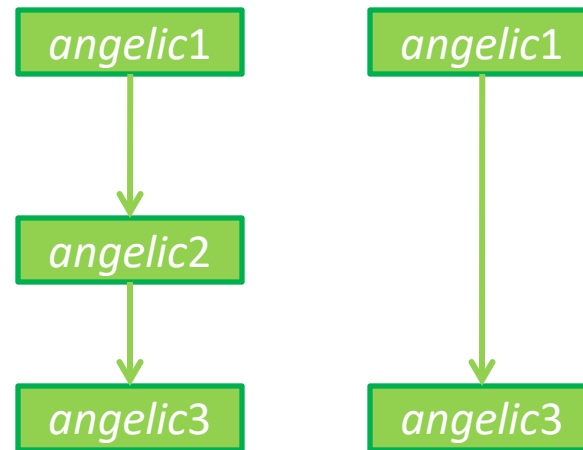


# Angelic Forest

Program



Angelic Paths



# Synthesis



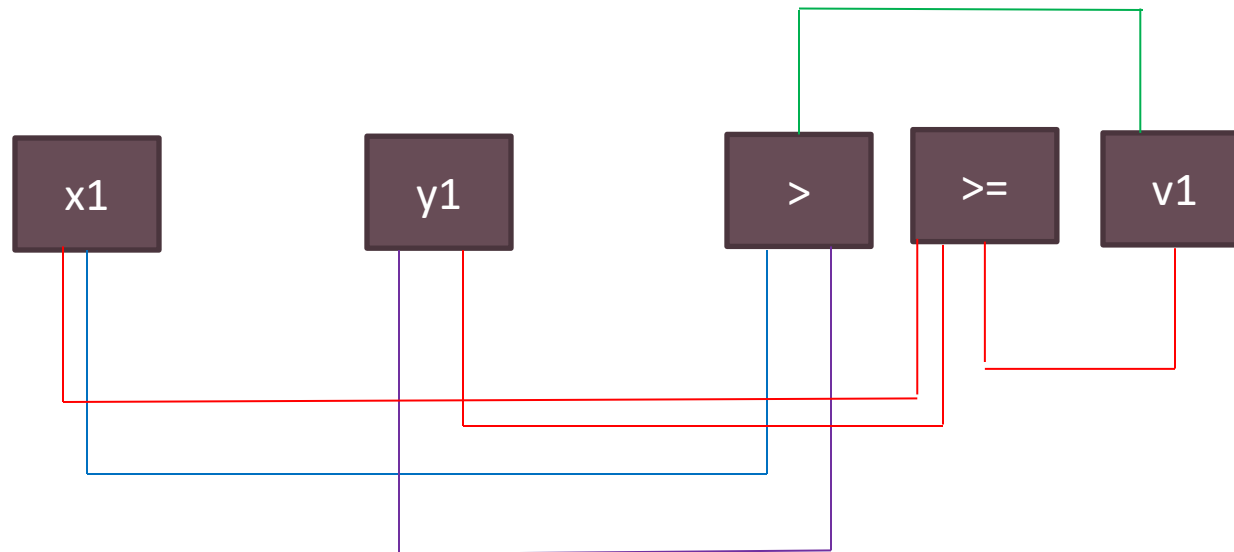
- Number of test cases is finite, therefore angelic forest is sufficient to specify expected behavior.
- Use repair synthesis as in DirectFix. For angelic forest  $\{\alpha_1, \alpha_2, \dots\}, \{\beta_1, \beta_2, \dots\} \dots$ , oracle constraints are:

$$(e_1 = \alpha_1 \wedge e_2 = \alpha_2 \dots) \vee (e_1 = \beta_1 \wedge e_2 = \beta_2 \dots) \dots$$

- Size of angelic forest is independent of the size of the program, and depends on the number of suspicious locations

# Repair via Soft Clause modification

$$L(>_1^{in}) = L(x1^{out}) \wedge L(>_2^{in}) = L(y1^{out}) \wedge L(>^{out}) = L(v1^{in})$$

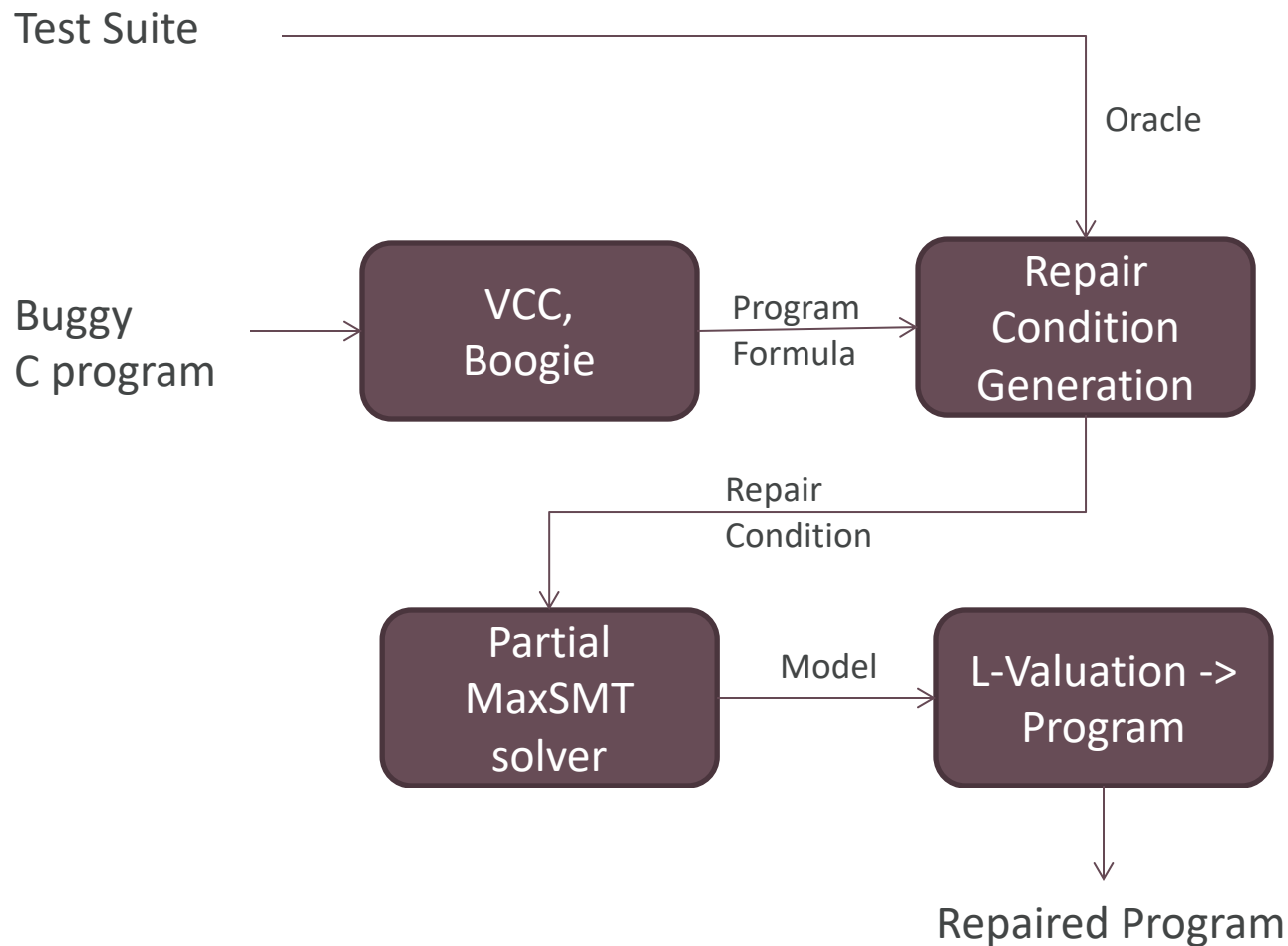


$$L(>=_1^{in}) = L(x1^{out}) \wedge L(>=_2^{in}) = L(y1^{out}) \wedge L(>=_^{out}) = L(v1^{in})$$

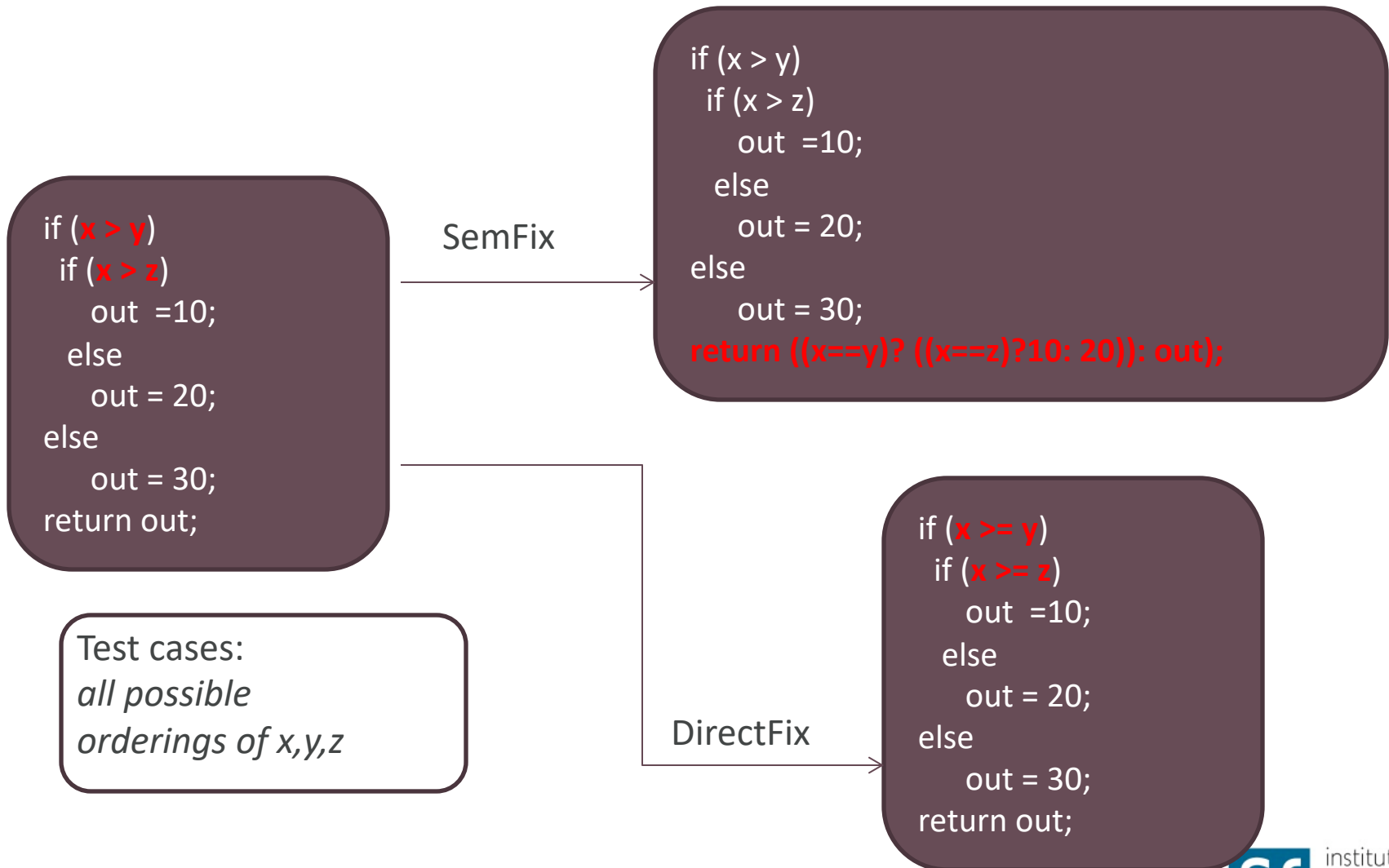
# How does the modification happen?

- Have a location variable for all components
  - Including those which are not used e.g.  $L_{>=}$
- MaxSMT solver removes some soft clauses
  - $L(>_1^{in}) = L(x1^{out})$   $L(>_2^{in}) = L(y1^{out})$   $L(>^{out}) = L(v1^{in})$
  - Remove  $L(>^{out}) = L(v1^{in})$
- Finds a model for the remaining formula
  - Valuation of L variables in the formula
  - Valuation of previously unconstrained variables  $L_{>=}$
- Model may show new connections  $L(>=^{out}) = L(v1^{in})$

# Implementation



# Example Program





# Heartbleed patch

```
if (hbtype == TLS1_HB_REQUEST
    && (payload + 18) < s->s3->rrec.length) {
    ...
} else if (hbtype == TLS1_HB_RESPONSE) {
    ...
}
return 0;
```

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0;
```

```
...
if (hbtype == TLS1_HB_REQUEST) {
    ...
} else if (hbtype == TLS1_HB_RESPONSE) {
    ...
}
return 0;
```

Generated patch



Developer patch

# Open problem: What is a high quality patch, anyway?

- Understandable?
  - Well, I had no problem understanding the POST-deleting patch...
  - (non-functional properties are important and being studied by others!)
- Doesn't delete?
  - But what about goto fail?
- Does the same thing the human did/would do?
  - But humans are often wrong! And how close does it have to be?
- Addresses the cause, not the symptom...

# Proposal: measure quality based on degree to which results *generalize*.

- In machine learning, techniques are trained and evaluated on disjoint datasets to assess overfitting.
- In program repair:
  - Tests used to build a repair are *training* tests
  - Tests used to assess correctness are *evaluation* tests

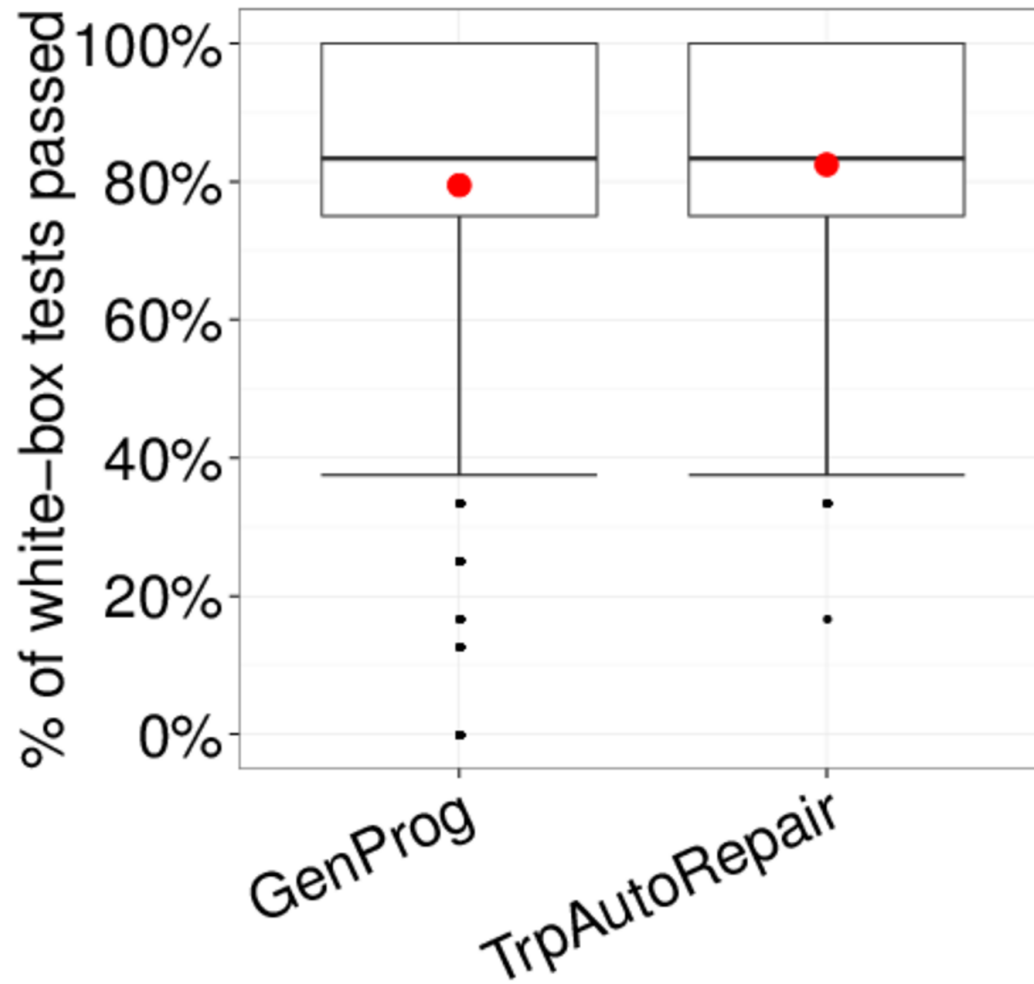


**PROBLEM: THE DESIRED STUDY IS IMPOSSIBLE.**

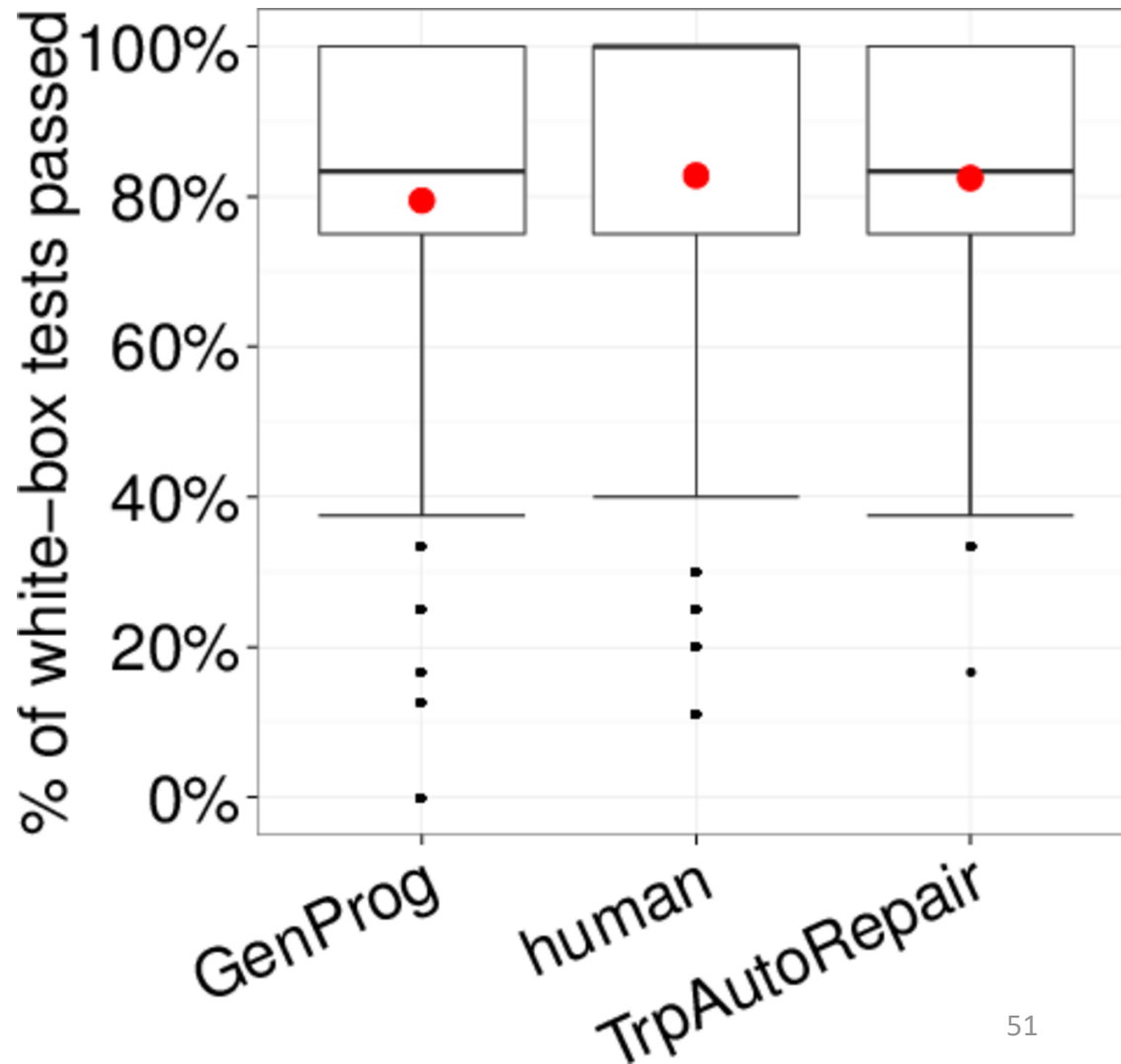
# [Dataset + Tools]

- Student homework submissions from six UC Davis Introduction to Programming assignments
- Two full-coverage test suites:
  - White-box suite generated by Klee from reference implementation.
  - Black-box suite written by course instructor.
  - *Feature: Assess patch quality as distinct from test suite quality.*
- Goal: Compare two different heuristic techniques to assess output quality.

# Both tools produced patches that overfit to the training set.



# But: the tools do as well as the students!



# Overfitting is not unique to heuristic techniques.

- Angelix: 120/233 of patches produced on a subset to IntroClass overfit.
- ~40% of SPR patches studied in Angelix paper delete functionality by generating tautological if conditions.



# Fast Forward to the 2019 ICSE SEIP Track....

## SapFix: Automated End-to-End Repair at Scale

A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, A. Scott

Facebook Inc.

“Facebook, Inc”

We report our experience with SAPFIX: the first end-to-end automated repair system for production code<sup>1</sup>. We have used SAPFIX to repair 6 production systems, each consisting of tens of millions of lines of code, and which are used by hundreds of millions of people worldwide.

### INTRODUCTION

Automated program repair seeks to find small changes to source code that patch the program [1]. One widely studied approach uses software testing to guide the repair process, as typified by the GenProg approach to search-based program repair [3].

[4], has been deployed at scale [5], [6]. The deployment of Sapienz allows us to find hundreds of crashes per month, before they even reach our internal human testers. Our software engineers have found fixes for approximately 75% of Sapienz-reported crashes [6], indicating a high signal-to-noise ratio [5] for Sapienz bug reports. Nevertheless, developers' time and expertise could undoubtedly be better spent on more creative programming tasks if we could automate some or all of the comparatively tedious and time-consuming repair process.

The deployment of Sapienz automated test design means that automated repair can now also take advantage of automated software test design to automatically re-test candidate patches. Therefore, we have started to deploy automated repair, in a tool called SAPFIX, to tackle some of these crashes. SAPFIX automates the entire repair life cycle end-to-end with the help of Sapienz: from designing the test cases that detect the crash, through to fixing and re-testing, the process is fully automated and deployed into Facebook's continuous integration and deployment system.

The Sapienz deployment at Facebook, with which SapFix integrates, tests Facebook's apps using automated search over the space of test input sequences [7]. This paper focuses on the deployment of SapFix, which has been used to suggest fixes for six key Android apps in the Facebook App Family, for which the Sapienz test input generation infrastructure has also been deployed. These are Facebook, Messenger, Instagram, FBLite, Workplace and Workchat. These six Android apps collectively consist of tens of millions of lines of code and are used daily by hundreds of millions of users worldwide to communicate, socialise and community building.

The first author, Alexandru Marginean, undertook the primary SAPFIX implementation work. The remaining authors contributed to the design, deployment and development of SAPFIX; remaining author order is alphabetical and not intended to denote any information about the relative contribution.

In order to deploy such a fully automated end-to-end detect-and-fix process we naturally needed to combine a number of different techniques. Nevertheless the SAPFIX core algorithm is a simple one. Specifically, it combines straightforward approaches to mutation testing [8], [9], search-based software testing [6], [10], [11], and fault localisation [12] as well as existing developer-designed test cases. We also needed to deploy many practical engineering techniques and develop new engineering solutions in order to ensure scalability.

SAPFIX combines a mutation-based technique, augmented by patterns inferred from previous human fixes, with a reversion-assist resort strategy for high-firing crashes (that would otherwise block further testing, if not fixed or removed). This core fixing technology is combined with Sapienz automated test design, Infer's static analysis and the localisation infrastructure built specifically for Sapienz [6]. SAPFIX is deployed on top of the Facebook FBLeaRner Machine Learning infrastructure [13] into the Phabricator code review system, which supports the interactions with developers.

Because of its focus on deployment in a continuous integration environment, SAPFIX makes deliberate choices to sidestep some of the difficulties pointed out in the existing literature on automated program repair (see Related Work section). Since SAPFIX focuses on null-dereference faults revealed by Sapienz test cases as code is submitted for review it can re-use the Sapienz fault localisation step [6]. The focus on null-dereference errors also means that a limited number of fix patterns suffice. Moreover, these particular patterns do not require additional fix ingredients (sometimes known as *donor code*), and can be applied without expensive exploration.

We report our experience, focusing on the techniques required to deploy repair at scale into continuous integration and deployment. We also report on developers' reactions and the socio-technical issues raised by automated program repair. We believe that this experience may inform and guide future research in automated repair.

The SAPFIX project is a small, but nevertheless distinct advance, along the path to the realisation of the FiFiVerify vision [10] of fully automated and verified code improvement. The primary contributions of the present paper, which reports on this deployment of SAPFIX are:

- 1) The first end-to-end deployment of industrial repair;
- 2) The first combination of automated repair with static and dynamic analysis for crash identification, localisation and re-testing;
- 3) Results from repair applied to 6 multi-million line systems;
- 4) Results and insights from professional developers' feedback on proposed repairs.

“one widely-studied [repair] approach uses software testing to guide the repair process, as typified by GenProg.”

“Results from repair applied to 6 multi-million line systems.”

# GenProg can't fix this, right?

- The checksum program should:
  - Take a single-line string as input.
  - Sum the integer codes of the characters, excluding the newline, modulo 64, plus the code for the space character.
- Buggy student assignment →

```
1
2
3
4
5
6
7
8
```

Incorrectly includes the newline in the sum.

```
    ... '\n' )
    ... "%c", &next );
    sum += next;
}
sum = sum % 64 + 22;
return sum;
```

Wrong value:  
the ASCII value  
of space is 32,  
not 22.

# Claire's favorite patch, ever

- The checksum program should:
  - Take a single-line string as input.
  - Sum the integer codes of the characters in the string, modulo 64, plus the code for the space character.
- GenProg fix with new representation →

```
1. // ...
2. while (next != '\n')
3. {
+   FIXME scanf("%c",
+             &next);
4.     sum += next;
+   if (next == '\n')
+       break;
4. }
5. sum = sum % 64 + 22;
+   sum += next;
8. return sum;
```



**Cool!**