COMP5111 – Fundamentals of Software Testing and Analysis

Automated Program Repair

Shing-Chi Cheung

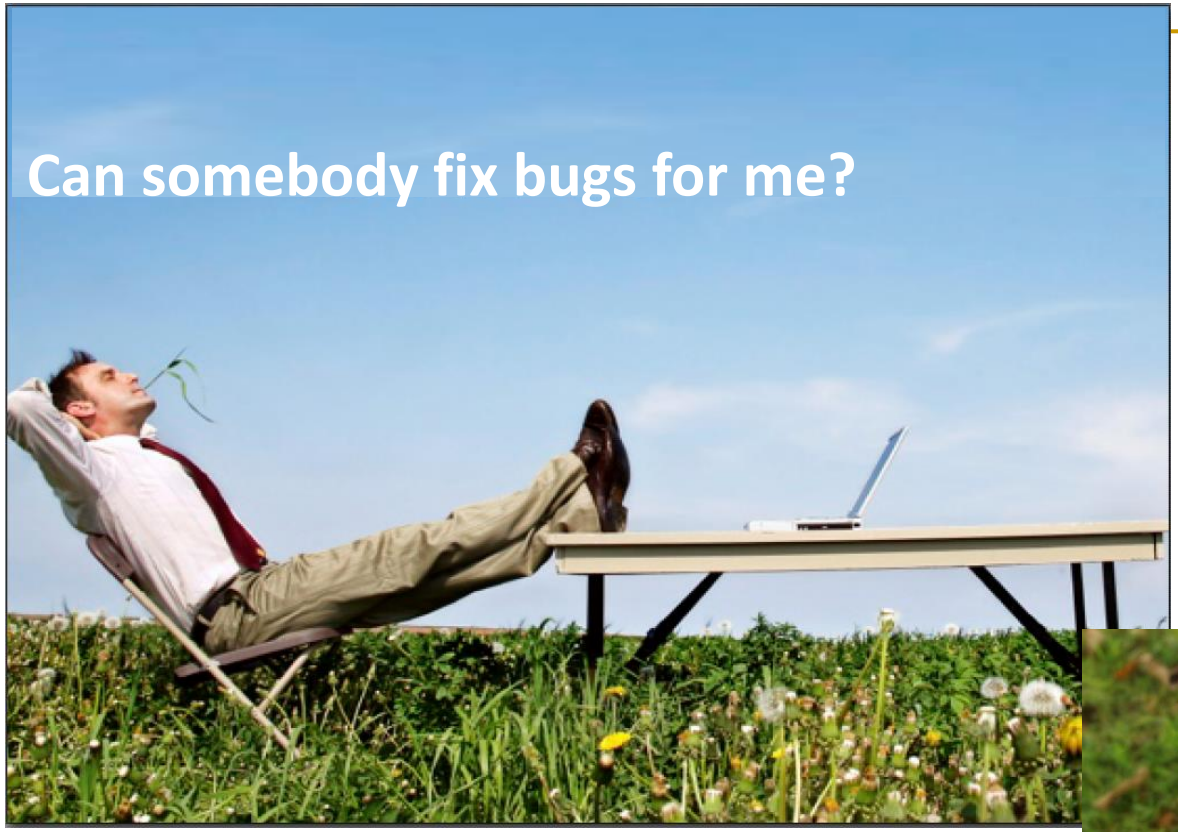
Computer Science & Engineering

HKUST

Adapted from the presentation slides by Ming Wen (2016)

Sound Familiar?

Done!



Dream of a developer…

MIT Technology Review

Log i

Topics+ The Down

Business Impact

A bot disguised as a human software developer fixes bugs

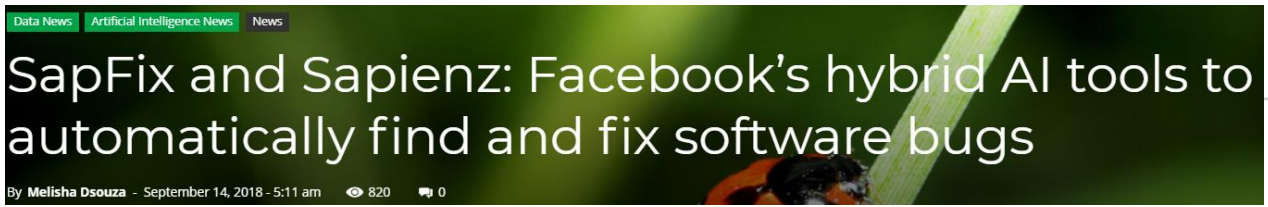
The automated programmer, called Repairnator, wrote patches good enough to fool actual human engineers.

by Emerging Technology from the arXiv October 23, 2018

"In this world nothing can be said to be certain, except death and taxes," wrote Benjamin Franklin in 1789. Had he lived in the modern era, Franklin may well have added "software bugs" to his list.

Tired of Debugging Code? Facebook's SapFix Tool Automates the Entire Process

PRANAV DAR, SEPTEMBER 14, 2018



Overview

- SapFix is a tool that automatically debugs your programming script
- It's intelligent enough to evaluate various bug fixes and provide a recommendation to the engineers
- Facebook plans to open source the tool in the near future once it has finished designing the finer details

Methodology

Automated Software Engineering

- The estimated cost of debugging is **US$312 billion** per year[1]
- Developers spend **50%** of their time debugging[1]

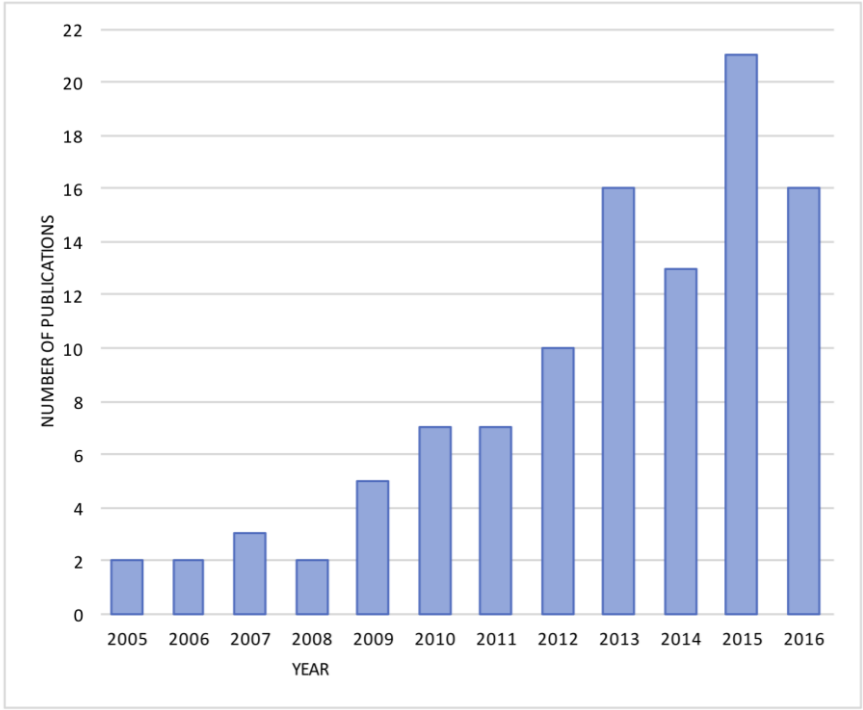
Four major bug diagnosis paradigms:

Automated Bug Prediction[2]	Automated Bug Detection ✓	Automated Bug Localization ✓	Automated Bug Repair?

1. Britton, Tom, et al. "Reversible debugging software." *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep* (2013).
2. Wen, Wu & Cheung. "How Well Do Change Sequences Predict Defects? Sequence Learning from Software Changes", TSE20.

Automated Program Repair (APR)

- The problem was first formulated in 2005 [Jobstmann, CAV05]
- Since GenProg [Weimer, ICSE09] was proposed in 2009, APR has received great research interest



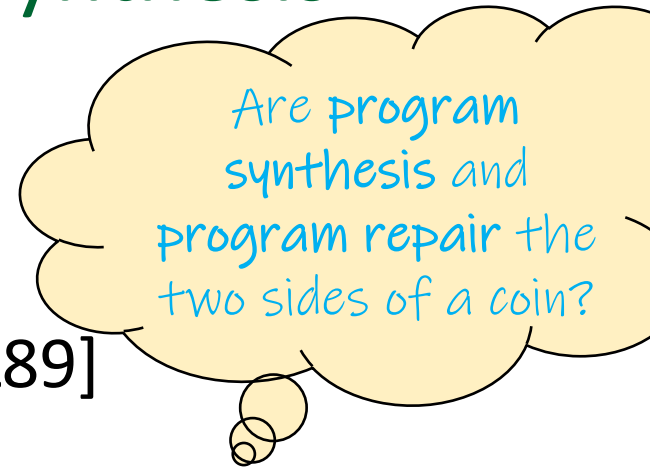
Source: Gazzola et al. Automatic Software Repair: A Survey, TSE 19.

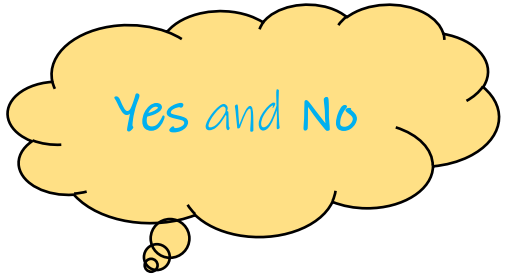
Motivation of Automated Program Repair

- Key empirical finding: Developers fix bugs much quicker if presented with high quality fix candidates
  - Yida Tao, Jindae Kim, Sunghun Kim, Chang Xu: Automatically generated fixes as debugging aids: a human study. SIGSOFT FSE 2014: 64-74

Program Repair for Program Synthesis

- Amir Pnueli (ACM Turing Award winner) stated that program synthesis is the hardest CORE problem in computer science [POPL89]
- Program synthesis
  = Program repair of an empty program
- Program repair can be a means to solve the program synthesis problem





Key Insight for Automated Program Repair

- Weimer found that code copying (6-40 tokens) is common among developers [ICSE 2009]
- It means developers often program to accomplish many small but similar tasks
  - Another implication is that developers are largely lazy in coding from scratch!
- Key insight: A piece of faulty code is likely to be fixed by another piece of similar code found elsewhere
- GenProg claimed to fix 55 out of 105 bugs [LeGoues et al., ICSE12]
  - It was later found to be an overclaim: Only 2 out of the 55 are correct fixes (i.e., precision at 3.63%, recall at 1.90%)

State-of-the-art Performance (up to 2018)

- CapGen [Wen et al., ICSE18] *// by HKUST CASTLE team*
  - Applied to 224 real Defect4J* bugs
  - Precision: 84%
  - Recall: 9.3%
- SimFix [Jiang et al., ISSTA18] *// by Peking University team*
  - Applied to 357 real Defect4J bugs
  - Precision: 60.7%
  - Recall: 9.52%
- Fix up to two statements at the same faulty location

Bug fixed by these two techniques are largely complementary

Altogether about 20% of bugs can be fixed automatically

Defects4J Dataset

Subject	#Bugs	KLOC
Commons **Lang**	65	22
JFree**Chart**	26	96
Commons **Math**	106	85
Joda-**Time**	27	28
☞ **Closure** Compiler	133	147
Total	357	378

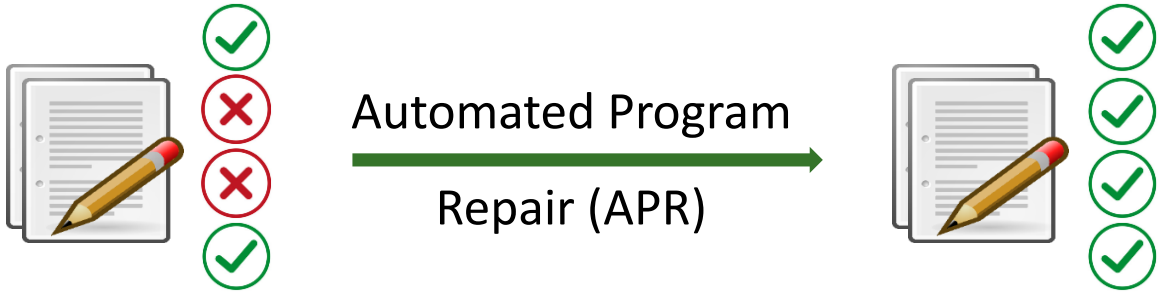
Now expanded to 10+ projects

- Compiler is not considered a regular application. Closure project is not commonly used for evaluating APR effectiveness

Core Concepts

- A program is buggy if its actual behavior deviates from its expected behavior
- In automated program repair, the expected behavior is usually specified by the test suite provided
- A program contains a bug when a test fails.

JUnit

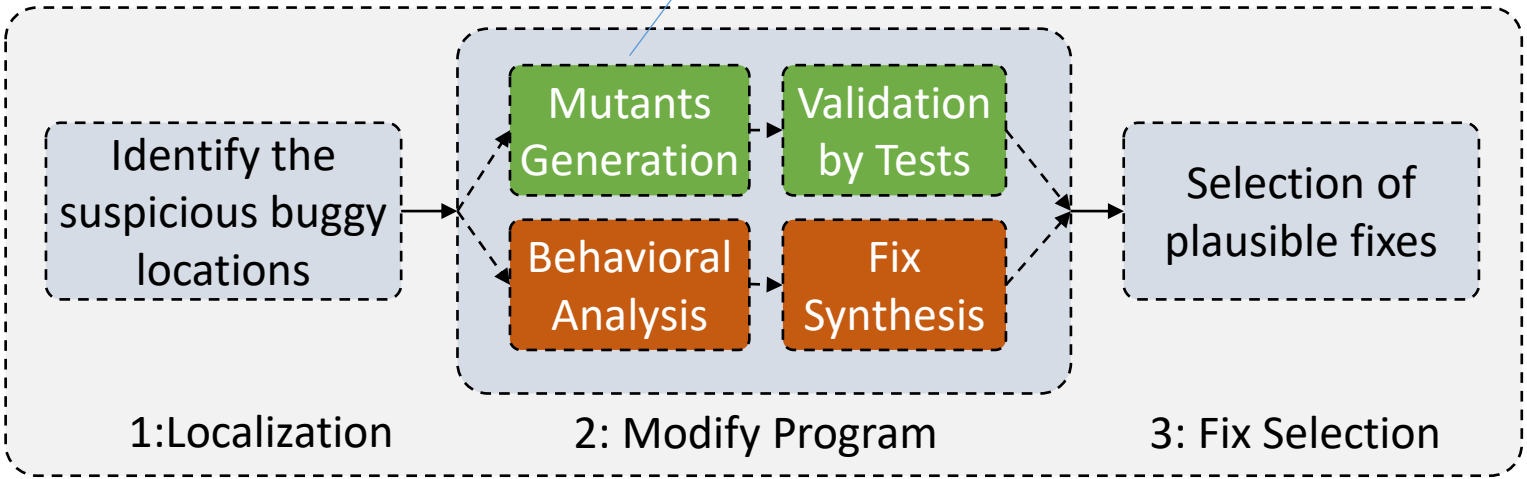


*[**Input:** Buggy Program + Test Outcomes]*

*[**Output:** A Fixed Program]*

General Process

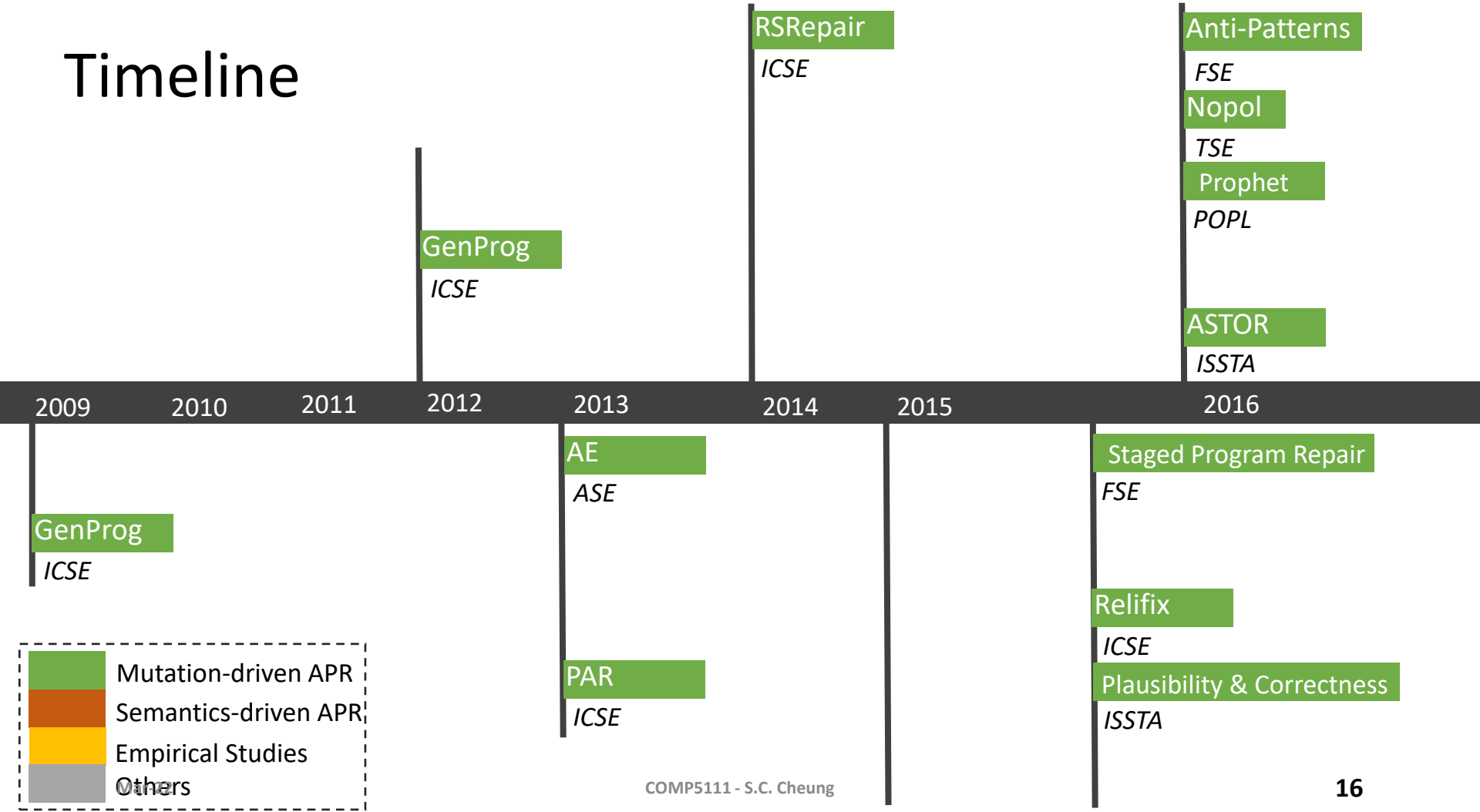
(also known as **generate-and-validate**)



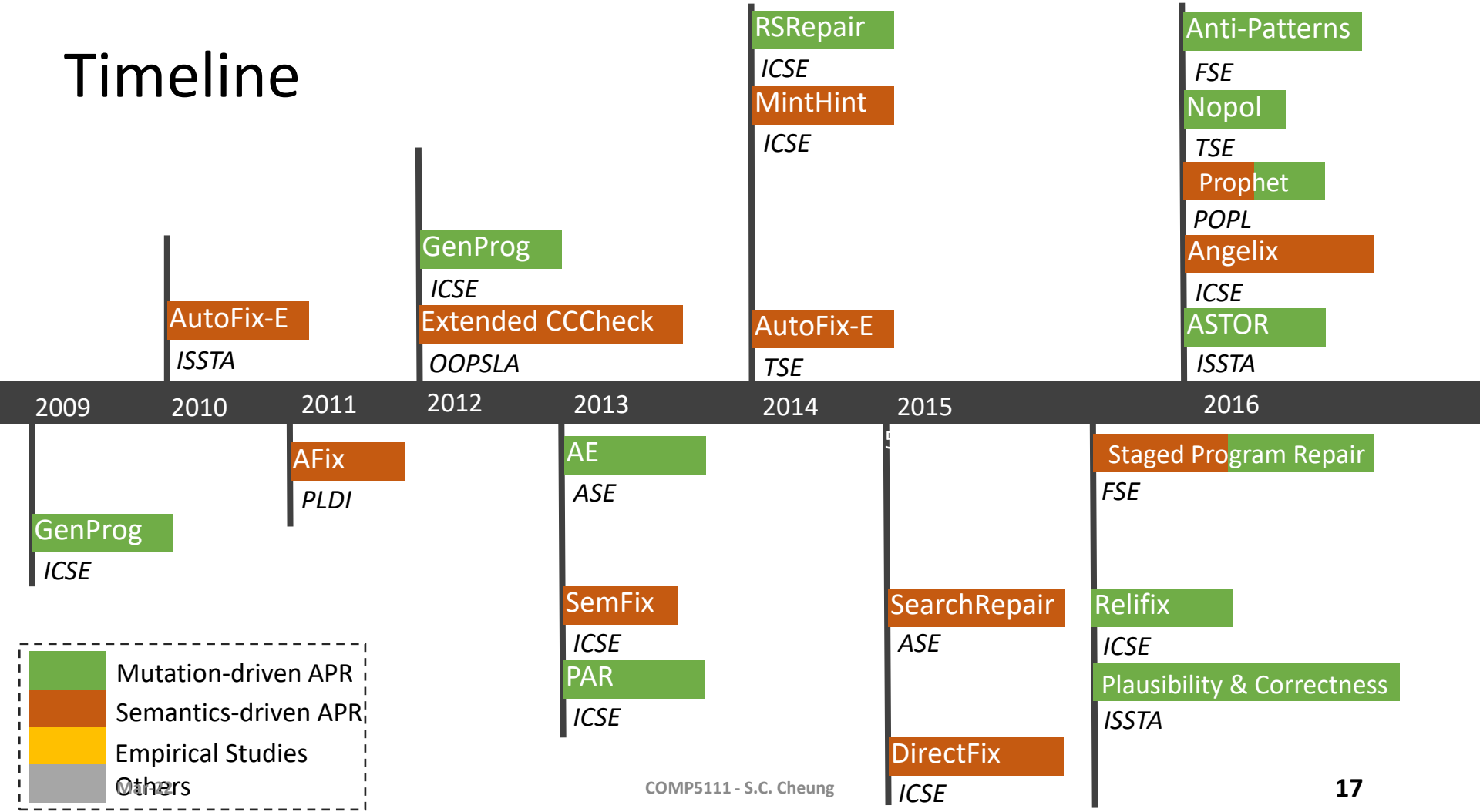
Mutation-driven ➡ Generate mutants

Semantics-driven ➡ Generate constraints

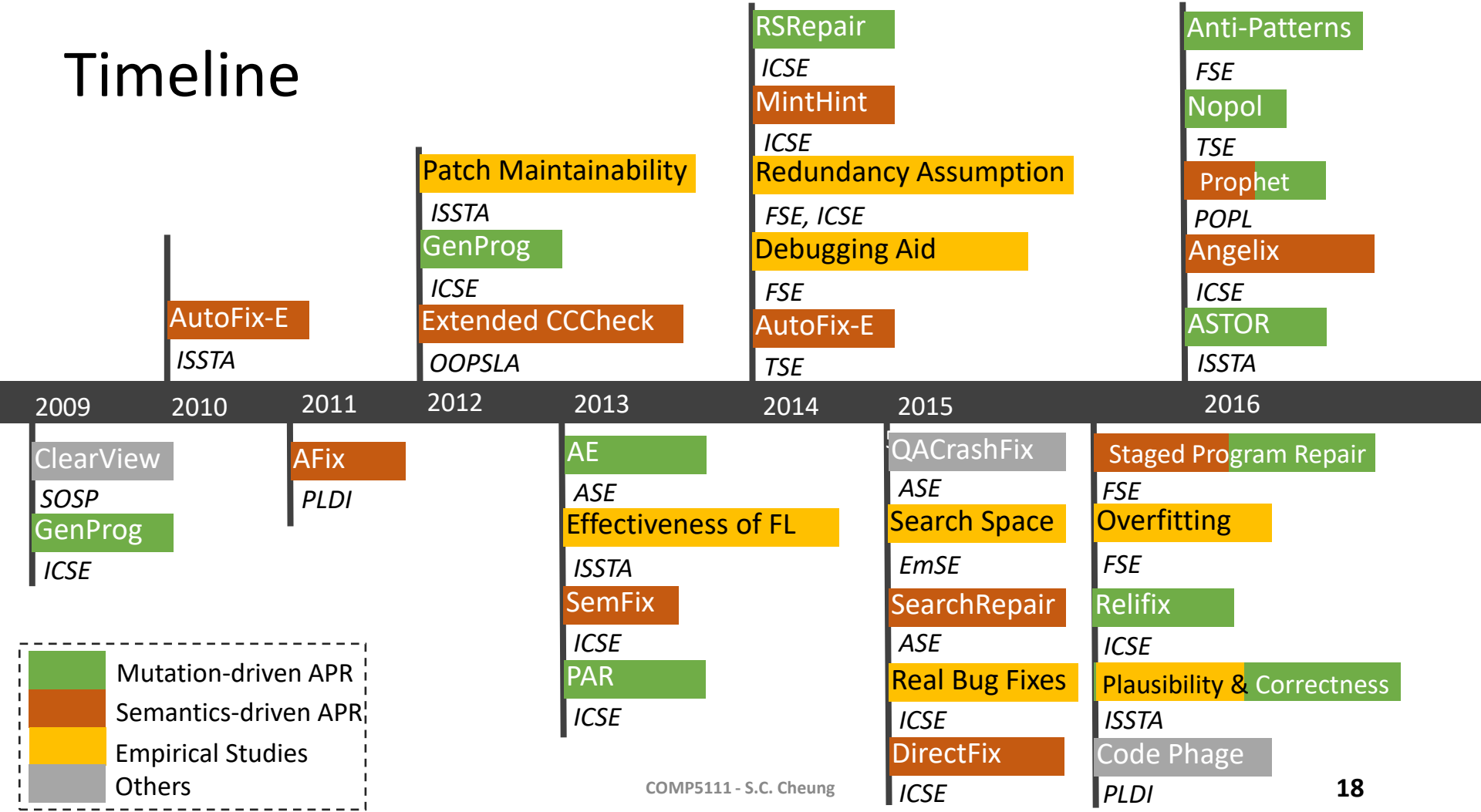
Timeline


RSRepair
ICSE
Anti-Patterns
FSE
Nopol
TSE
Prophet
POPL
GenProg
ICSE
ASTOR
ISSTA
2009
2010
2011
2012
2013
2014
2015
2016
AE
ASE
Staged Program Repair
FSE
GenProg
ICSE
Relifix
ICSE
PAR
ICSE
Plausibility & Correctness
ISSTA
Mutation-driven APR
Semantics-driven APR
Empirical Studies
Others

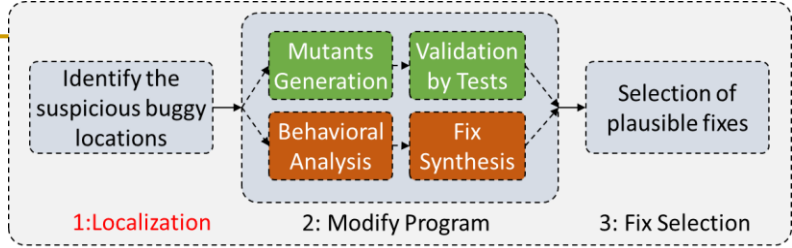
Timeline


2009
2010
2011
2012
2013
2014
2015
2016
GenProg
ICSE
AutoFix-E
ISSTA
AFix
PLDI
GenProg
ICSE
Extended CCCheck
OOPSLA
AE
ASE
SemFix
ICSE
PAR
ICSE
RSRepair
ICSE
MintHint
ICSE
AutoFix-E
TSE
SearchRepair
ASE
DirectFix
ICSE
Anti-Patterns
FSE
Nopol
TSE
Prophet
POPL
Angelix
ICSE
ASTOR
ISSTA
Staged Program Repair
FSE
Relifix
ICSE
Plausibility & Correctness
ISSTA
Mutation-driven APR
Semantics-driven APR
Empirical Studies
Others

Timeline


2009
2010
2011
2012
2013
2014
2015
2016
ClearView
SOSP
GenProg
ICSE
AutoFix-E
ISSTA
AFix
PLDI
Patch Maintainability
ISSTA
GenProg
ICSE
Extended CCCheck
OOPSLA
AE
ASE
Effectiveness of FL
ISSTA
SemFix
ICSE
PAR
ICSE
RSRepair
ICSE
MintHint
ICSE
Redundancy Assumption
FSE, ICSE
Debugging Aid
FSE
AutoFix-E
TSE
QACrashFix
ASE
Search Space
EmSE
SearchRepair
ASE
Real Bug Fixes
ICSE
DirectFix
ICSE
Anti-Patterns
FSE
Nopol
TSE
Prophet
POPL
Angelix
ICSE
ASTOR
ISSTA
Staged Program Repair
FSE
Overfitting
FSE
Relifix
ICSE
Plausibility & Correctness
ISSTA
Code Phage
PLDI
Mutation-driven APR
Semantics-driven APR
Empirical Studies
Others

Spectrum-Based Fault Localization



- The basic idea is to statistically analyze the dynamic execution information to locate bugs.

Program

Test Suite

	T_1	T_2	T_3	T_4	T_5
S_1	✓	✓	✓	✓	✓
S_2					✓
...					
S_{n-1}		✓	✓	✓	✓
S_n	✓	✓	✓		

Program Profiling Information

ranking function

Entity	Suspicious
S_1	0.50
S_2	1.00
...	
S_{n-1}	0.57
S_n	0.00

Suspicious Score

Fault Localization Ranking Function in APR

Table 1: Common Fault Localization Ranking Function Used in APR

Name	*Model*	*APR Techniques that Adopt it*
FaultLoc	$Susp(S) = \begin{cases} 0, if\ N_{SS} = 0\ and\ N_{FS} = 0 \\ 1, if N_{SS} = 0\ and\ N_{FS} > 0 \\ 0.1, otherwise \end{cases}$	**GenProg** [ICSE 2009, ICSE 2012] **AE** [ASE 2013] **PAR** [ICSE 2013] **RSRepair** [ICSE 2014]
Tarantula	$Susp(S) = \frac{N_{FS}/N_F}{N_{FS}/N_F + N_{SS}/N_S}$	**SemFix** [ICSE 13] **MutRepair** [ICSTVV 2010]
Ochiai	$Susp(S) = \frac{N_{FS}}{\sqrt{N_F * (N_{FS} + N_{SS})}}$	**SearchRepair** [ASE 2015] **Relifix** [ICSE 2016] **ASTOR** [ISSTA 2016] **Nopol** [TSE 2016]
Jaccard	$Susp(S) = \frac{N_{FS}}{N_F + N_{SS}}$	**Angelix** [ICSE 2016]

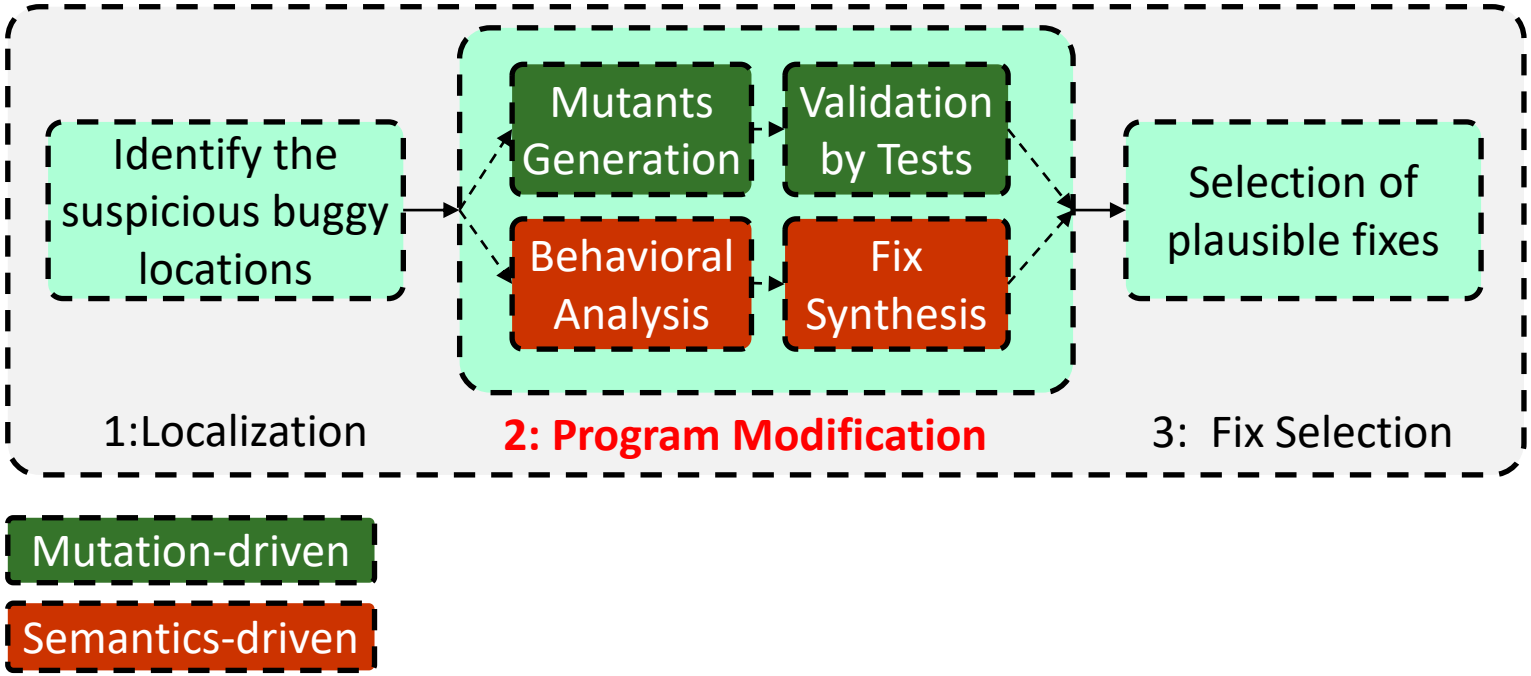
N_{FS} : The number of *failing* test cases that have executed statement S.
N_F : The total number of *failing* test cases.
N_{SS} : The number of *passing* test cases that have executed statement S.
N_S : The total number of *passing* test cases.

Fault Localization in APR

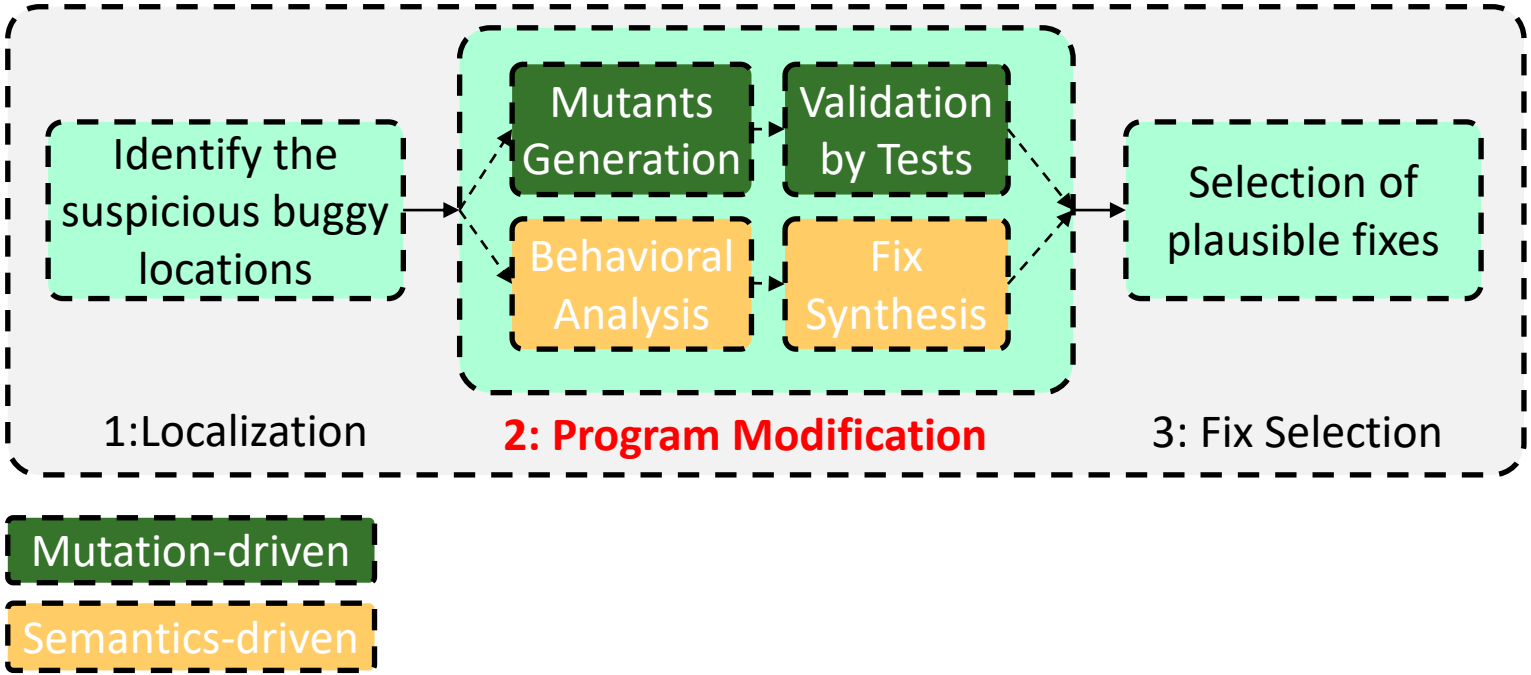
- Qi et al. [ISSTA 2013] evaluated the effectiveness of these fault localization algorithms in automated program repair.
- They applied 15 different algorithms in GenProg.
- Spectrum-based FL techniques perform well in APR in general.
- Jaccard outperforms other algorithms.

Ochiai is more popularly used by recent studies

Automated Program Repair (APR)



Automated Program Repair (Mutation-driven)

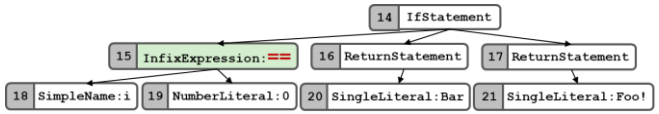


Mutation-driven APR

```
…
if (i == 0) return "Bar";
 else return "Foo!";
…
```

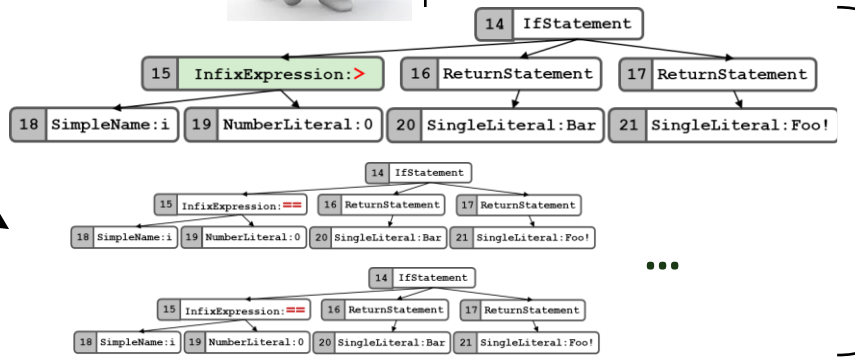
Suspicious statement

AST conversion



Random-based
Search-based
Rule-based

Mutate AST



Many mutants

Test validation

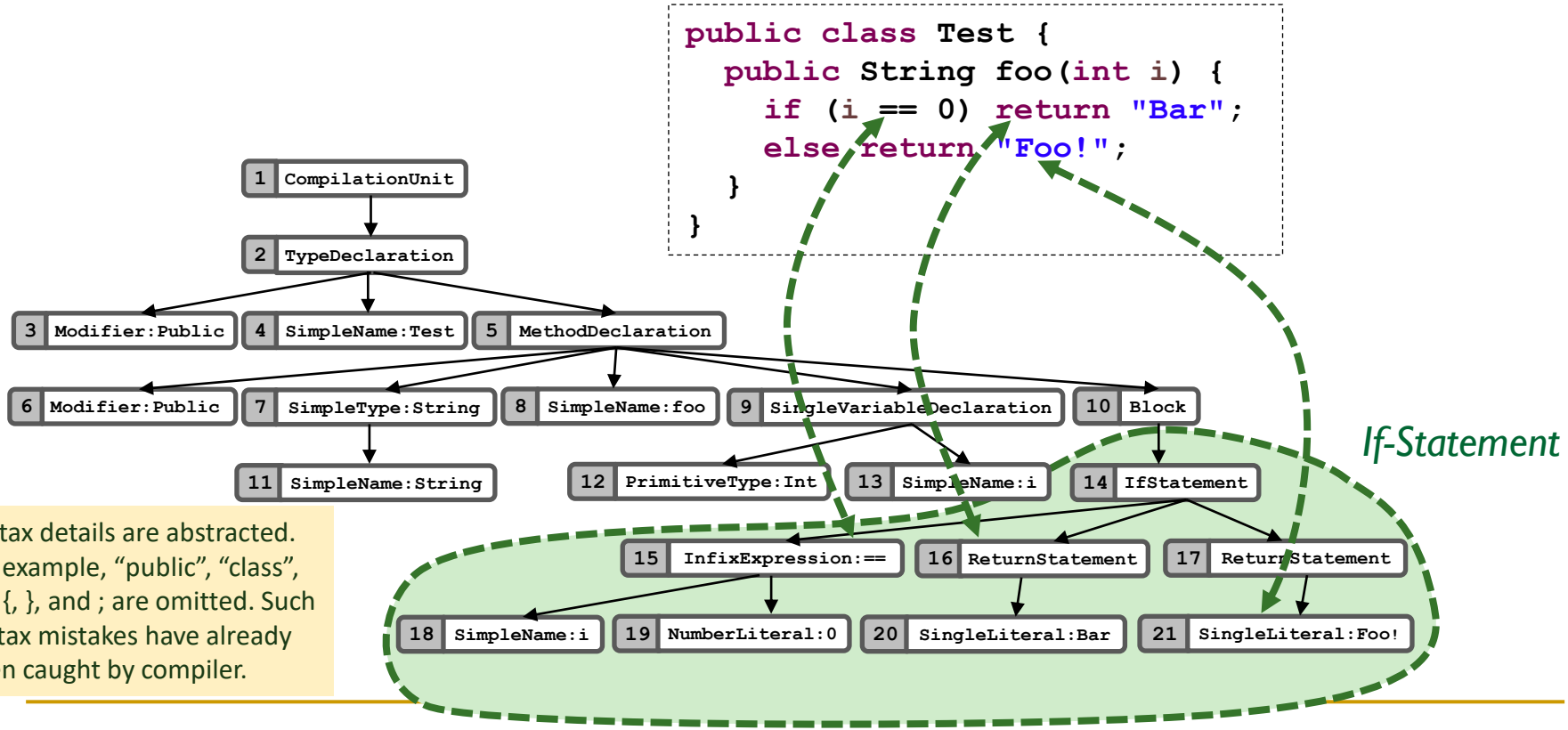
Plausible Fixes

Fix Priorization

Heuristics
Machine learning

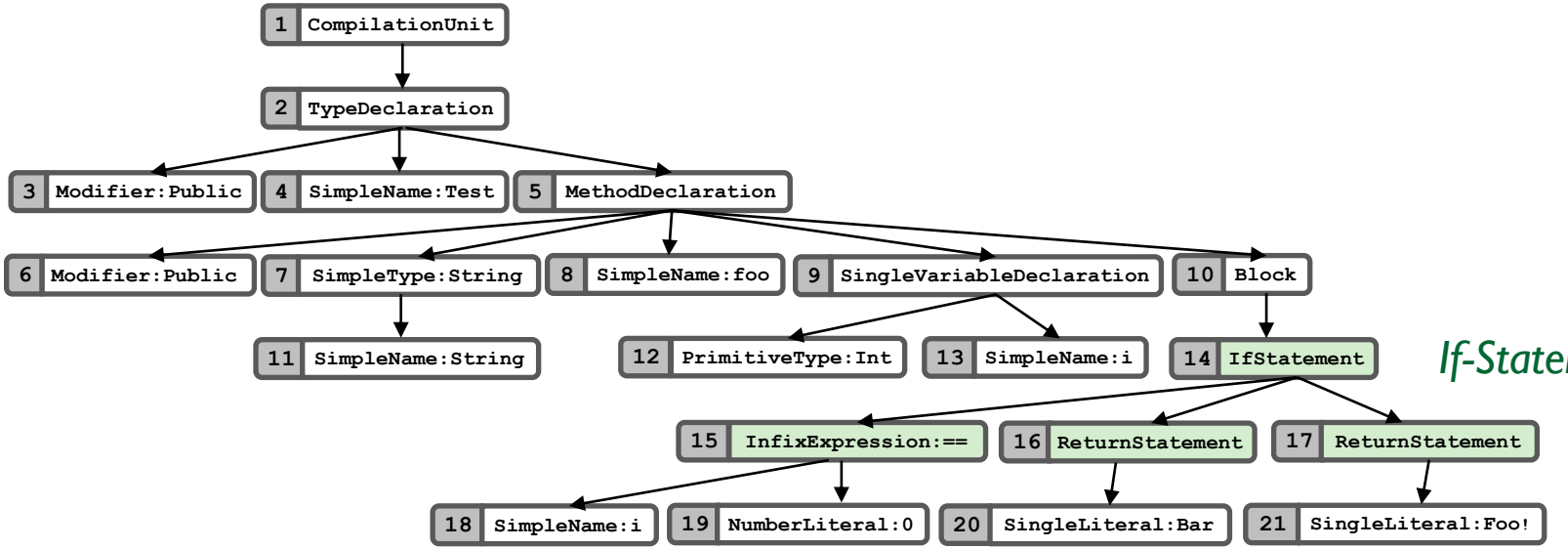
```
…
if (i >= 0) return "Bar";
 else return "Foo!";
…
```

Abstract Syntax Tree (AST)



Syntax details are abstracted. For example, “public”, “class”, (,), {, }, and ; are omitted. Such syntax mistakes have already been caught by compiler.

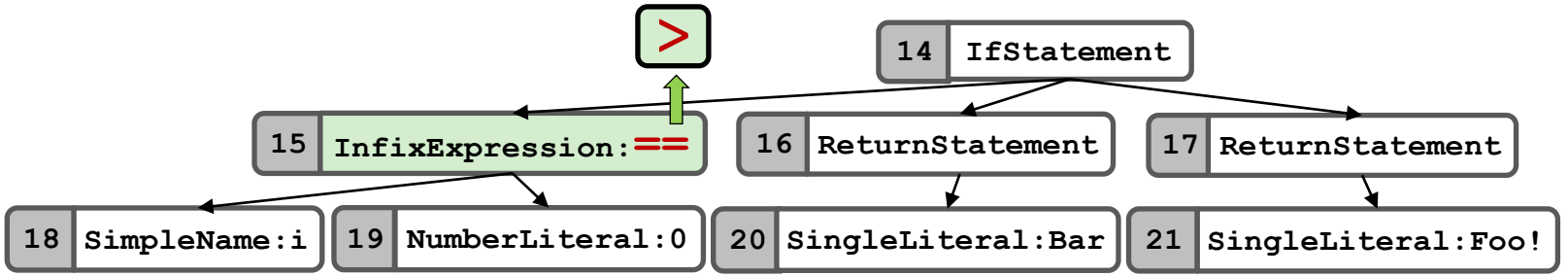
Abstract Syntax Tree (AST)



- There are 86 distinct node types defined in JDT (Java Defined Types)
- APR focuses on the node types related to Expression and Statement
- AST manipulation tools: CLI (for C) or Spoon (for Java)

Mutant Generation

- Generate modified programs by altering the suspicious statement using mutation operators

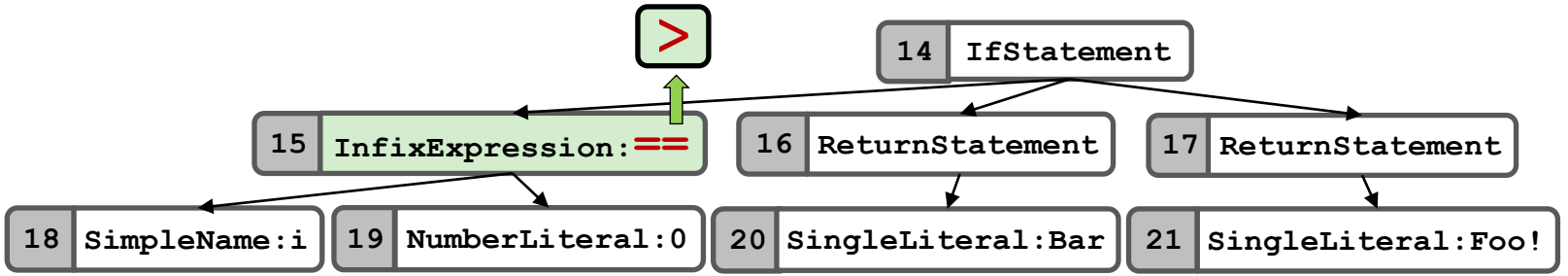



```
public class Test {
  public String foo(int i) {
    if (i == 0) return "Bar";
    else return "Foo!";
  }
}
```

```
public class Test {
  public String foo(int i) {
    if (i > 0) return "Bar";
    else return "Foo!";
  }
}
```

Mutant Generation

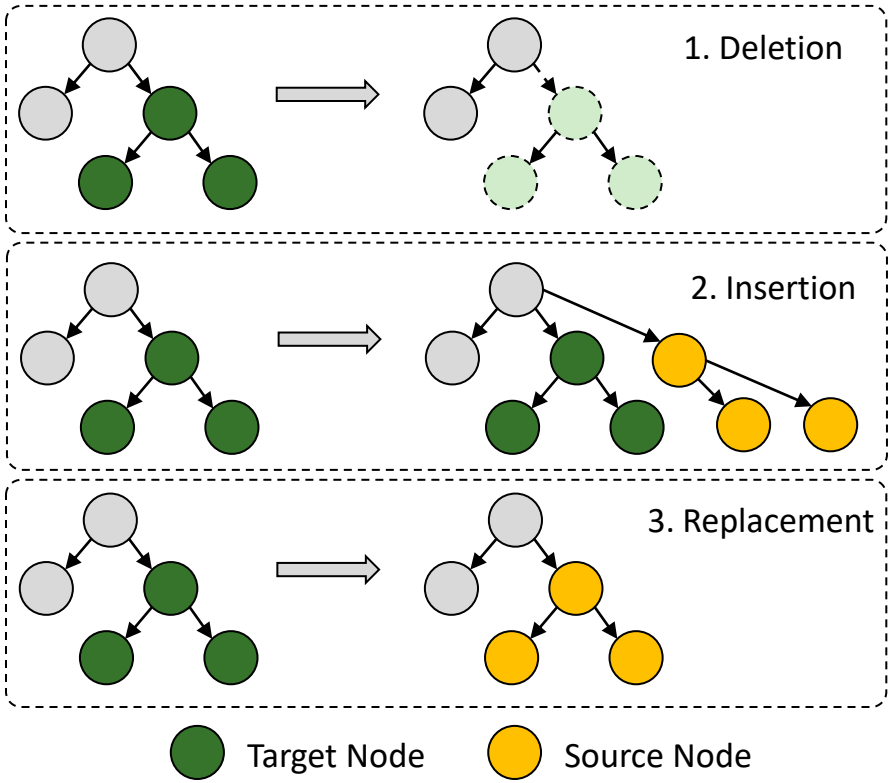
- Question: Why mutating AST?




```
public class Test {
  public String foo(int i) {
    if (i == 0) return "Bar";
    else return "Foo!";
  }
}
```

```
public class Test {
  public String foo(int i) {
    if (i > 0) return "Bar";
    else return "Foo!";
  }
}
```

Mutation Operators

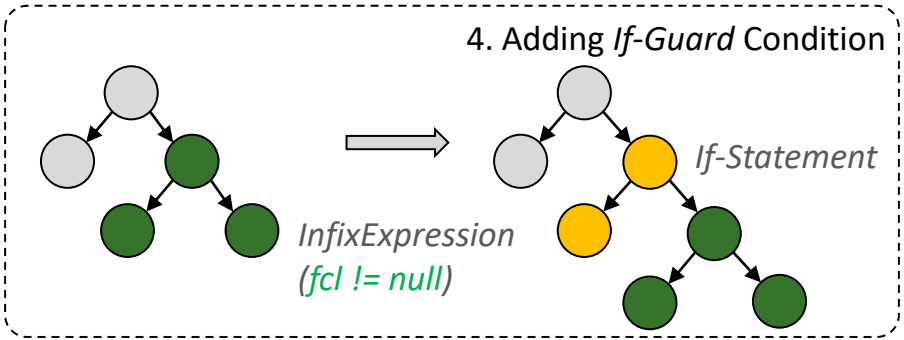


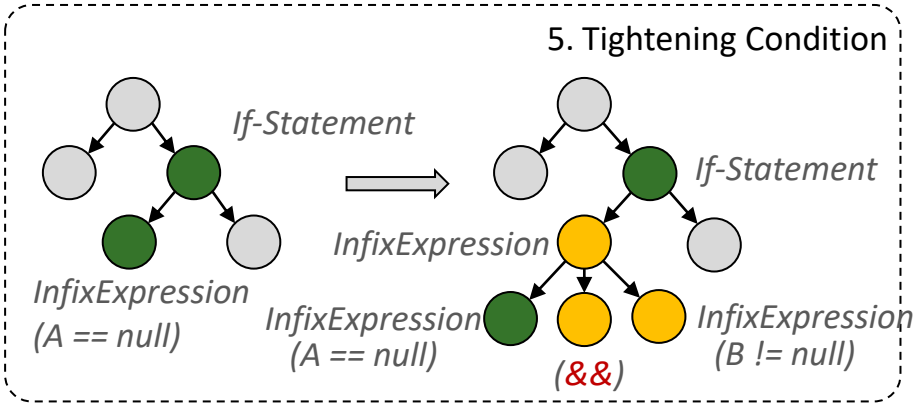
Redundancy Assumption
[Qi et al., ICSE14, Tao et al., FSE14]

The fix ingredients already exist elsewhere in the code.

- The same class *[Debroy et al., ICSTVV10, Martinez et al., ICSE14]*
- The same application *[LeGoues et al., ICSE12, Weimer et al., ASE13, DeMarco et al., ISSTA14 , Long et al., ESEC/FSE15, Long et al., POPL16]*
- Other applications *[Sidiroglou-Douskos et al., PLDI15]*

Mutation Operators






```
11      project.getProject();
12      ....
13  +    if (fcl != null) {
14          removeFileListener(fcl);
15  +    }
16
```

Many real bug fixes are related to buggy if conditions [EmSE 2009].

```
11 +  if (A == null && B != null){
12      result = annotated;
13    } else {
15      result = getResult();
16    }
```

Loosening condition can be achieved in the same way by replacing the operator && with ||.

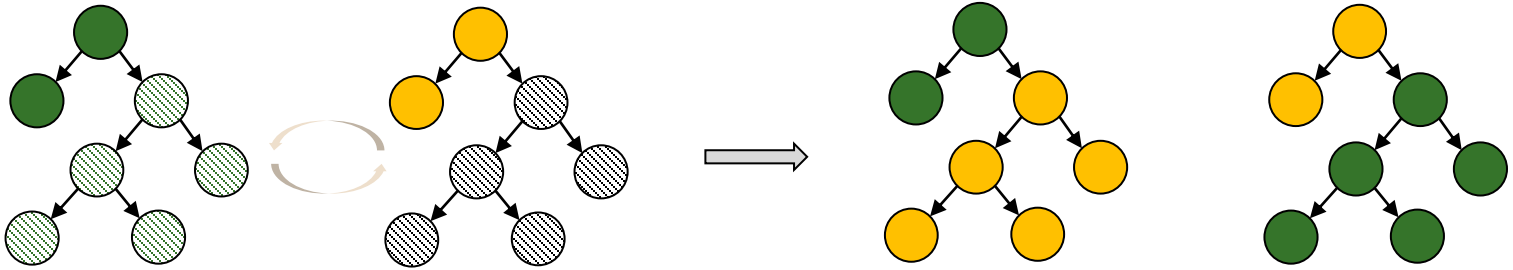
Mutation Operators

Table 2: Commonly Used Mutation Operators in Mutation-driven APR

Name	*GenProg*	*MutRepair*	*PAR*	*AE*	*RSRepair*	*SPR*	*Prophet*	*Kali*	*Nopol*	*ASTOR*
Insert	✓			✓	✓					✓
Replace	✓	✓		✓	✓	✓	✓			✓
Delete	✓			✓	✓			✓		✓
Tighten Condition			✓			✓	✓		✓	
Loosen Condition			✓			✓	✓		✓	
Negate Condition		✓	✓						✓	✓
Add Guard						✓	✓		✓	
Add Initialization			✓			✓	✓			

Patch Generation - GenProg

- Mutation Operator: *Insertion*, *Replacement*, and *Deletion*.
- Crossover Operator: Cross over between two candidates.



Patch Generation - GenProg

Genetic Programming

- Generate and select a large set of fix candidates by applying *mutation* and *crossover* operators
- Select fixes using the fitness function

$$Fitness(P) = W_{negT} * Num_{negT} + W_{\mathrm{pos}T} * Num_{posT}$$

- Performance
  - It was said to be capable of fixing *55* out of the *105* real-world bugs
  - Later found to have correctly fixed only *2* bugs!
  - Documented test suites of open-source projects are often weak

Limitation I - *Overfitting*

- When a test suite is weak, it cannot discriminate *correct fixes* from those *overfitting* to the tests
- GenProg suffers severely from overfitting [Long et al., ESEC/FSE15]
- Qi et al. found that many suggested fixes are only *function deletion* [ISSTA15].

Deletion

if (*condition*) return

instead of executing the faulty code caused by a specific test inputs

Weak Test Suite does not assure all desirable functionalities

Simply deleting functionalities (i.e., pruning the AST).

Limitation II - *Search Space Explosion*

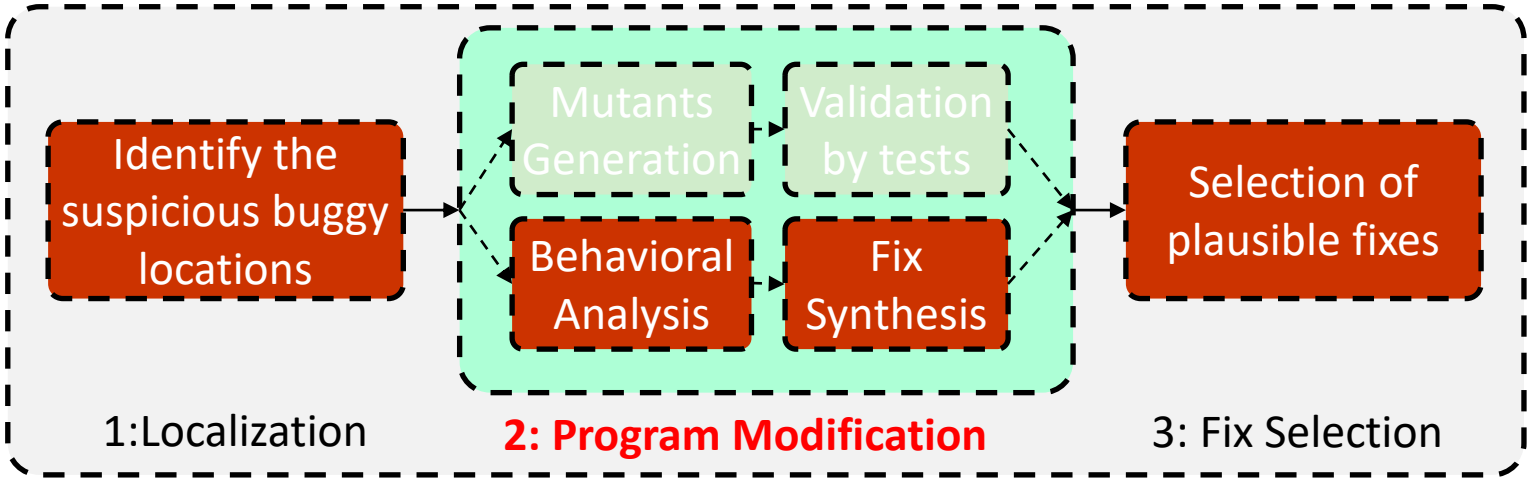
- The correct fixes are sparse in the huge search space [Mechtaev et al., ICSE16].
- Plausible fixes are typically orders of magnitude outnumber correct fixes [Mechtaev et al., ICSE16].

A fix is *plausible* if it passes the whole test suite

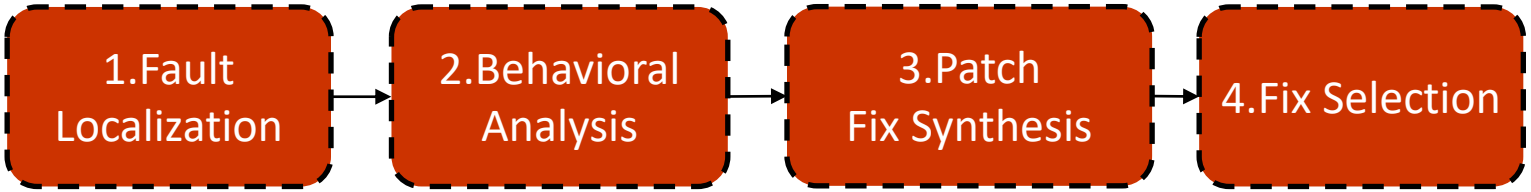
#Suspicious Statements ✖ *#Mutation Operators* ✖ *#Combination of Arguments for Mutation Operators*

Source Ingredients Space

Semantics-driven Program Repair (APR)



Semantics-driven APR



The General Steps of Semantics-driven APR

- Infer constraints by behavioral analysis
- Synthesize fixes by constraint solving
- No test validation is needed

What Kinds of Patches to Synthesize?

- *Conditions*
  - SPR [Long et al., ESEC/FSE15], Prophet [Long et al., POPL16], SemFix [Nguyen et al., ICSE13], Nopol [DeMarco et al., TSE16], DirectFix [Mechtaev et al., ICSE15], Angelix [Mechtaev et al., ICSE16]
- *Right hand side of an assignment*
  - SemFix [Nguyen et al., ICSE13], DirectFix [Mechtaev et al., ICSE 2015], Angelix [Mechtaev et al., ICSE16]
- *Function parameters*
  - DirectFix [Mechtaev et al., ICSE15], Angelix [Mechtaev et al., ICSE16]

Condition Fix Synthesis – SPR [ESEC/FSE15]

- Tighten/Loosen a condition.
  - if (cond) → if (cond && abstract_cond())
- Add an if-guard condition for a statement.
  - statement; → if (abstract_cond()) statement;
- ⇨ Add a potentially guarded control statement.
  - s1;s2; → s1; if (abstract_cond()) return; s2;

Condition Fix Synthesis – SPR [ESEC/FSE15]

```
void foo(...) {
   ...
   char *buf = malloc(nsize);


   memcpy(buf, a, nsize);
   ...
}
```

⇐ Suspicious statement

Add a potentially guarded control statement

- Create a template fix with an *abstract condition*
- Try different values of the *abstract condition* during the symbolic execution (initially set to 0) Flip the value when the execution failed.
- Record the *variable values* at each condition invocation(*buf, nsize*)
- Synthesize a concrete fix using the recorded variables

Condition Fix Synthesis – SPR [ESEC/FSE15]

```
void foo(...) {
    ...
    char *buf = malloc(nsize);
    if (_abst_cond())
        return;
    memcpy(buf, a, nsize);
    ...
}
```

	_abst_cond()	buf	nsize	
1st	0	0x30aa...	100	
2nd	0→1	0	100000	F→P
3rd	0	0x30ab...	200	
4th	0→1	0	200000	F→P

Add a potentially guarded control statement

Plausible fixes:

_abst_cond() → *buf == 0* ✓

_abst_cond() → *nsize >= 10000*

SPR fixes 38 bugs out of 69, while **GenProg** fixes only 16, **AE** fixes 25.

RHS of Assignment – SemFix [ICSE13]

- An *expression* is treated as a *function*
- The repair candidates are in one of the two forms:
  - $x = F_{buggy}(\dots) \rightarrow x = F_{fixed}(\dots)$
  - $if\ (F_{buggy}(\dots)) \rightarrow if\ (F_{fixed}(\dots))$
- The function takes all the *accessible variables* as parameters

RHS of Assignment – SemFix [ICSE13]

```
1  int G(int a, int b, int c) {
2   int bias;
3   if (a)
4    bias = c;
5   else
6    bias = b;
7   if (bias > c)
8    return 1; // a > 0 ∧ F(a,b,c) > c
9   else return 0; // a > 0 ∧ F(a,b,c) ≤ c
10 }
```

$bias = F(a,b,c)$

symbolic execution of suspicious statement

Test	Inputs			Expected output	Observed output	Status
	a	b	c			
1	1	0	100	0	0	Pass
2	1	11	110	1	0	Fail
3	0	100	50	1	1	Pass
4	1	-20	60	1	0	Fail
5	0	0	10	0	0	Pass

Test 1 Test 2 Test 4 …

$< 1,0,100 >< 1,11,110 >< 1,-20,60 >$ …

$F(a,b,c) = b + 100$ ⇦ $bias = F(1,0,100) \leq 100 \wedge F(1,11,110) > 110 \wedge F(1,-20,60) > 60$

Condition & RHS Assignment Patches - SemFix

The level of basic components for synthesizing

Decreasing order of preferences

Level	Conditional Statement	Assignment Statement
1	Constants	Constants
2	Comparison $(>, \geq, =, \neq)$	Arithmetic $(+, -)$
3	Logic (and, or)	Comparison
4	Arithmetic $(+, -)$	Logic
5	Array Access	Array Access
6	Arithmetic $(*)$	Arithmetic $(*)$

$bias = b + 100$ ⇦ $bias = F(1,0,100) \leq 100 \wedge F(1,11,110) > 110 \wedge F(1,-20,60) > 60$

Function Parameter Fix – DirectFix [ICSE15]

The constraint solving mechanism is similar to SemFix with enhancement of:

- The synthesized fixes should not only resolve the bug but also should keep certain readability and comprehensibility in order to be accepted by developers
  - Observation: Real fixes are readable and comprehensible
  - Hypothesis: A fix is more likely to be correct if it is readable and comprehensible
- DirectFix focuses on simplifying the fixes while preserving the program structure of the synthesized fixes
- Limitation: Can only be applied to relatively small programs

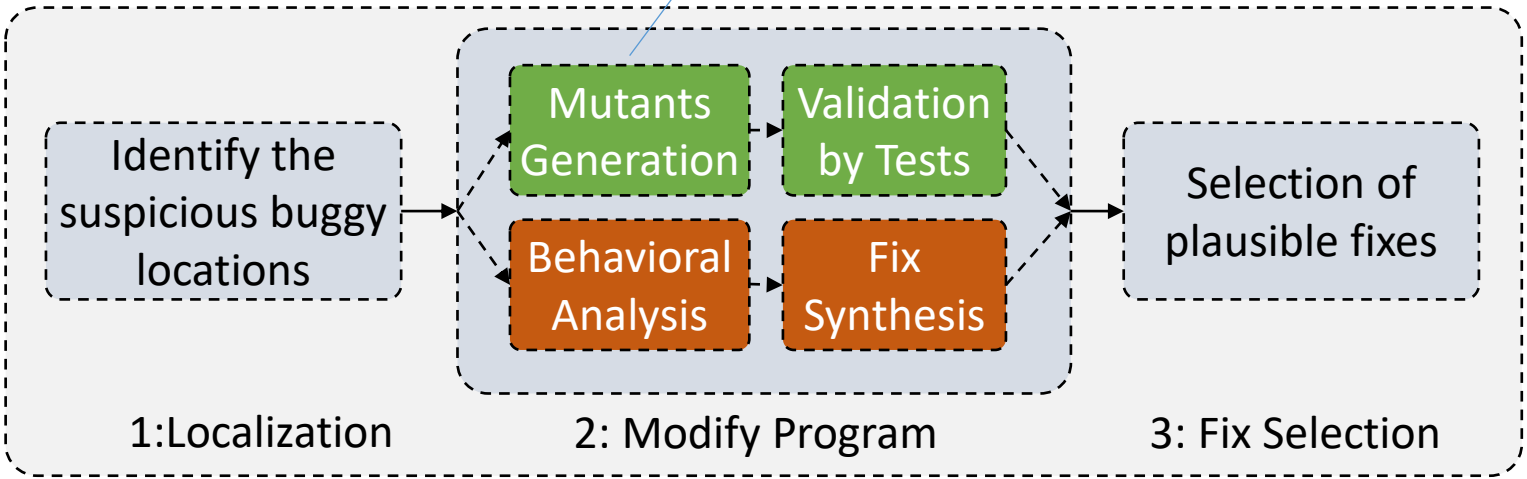
Limitations

Test	Inputs			Expected output	Observed output	Status
	a	b	c			
1	1	0	100	0	0	Pass
2	1	11	110	1	0	Fail
3	0	100	50	1	1	Pass
4	1	-20	60	1	0	Fail
5	0	0	10	0	0	Pass

- *Effectiveness*
  - The effectiveness of current techniques is *limited* to the capability of constraint solving and program synthesis
- *Scalability*
  - May fail to synthesize a correct fix in 12 hours due to the huge search space and complexity in constraint solving
- *Overfitting*
  - The repair fix not generalizable to other test cases not in the test suite
  - The inadequate test suites may render program synthesis to generate many incorrect plausible fixes

AI Approach - Emerging

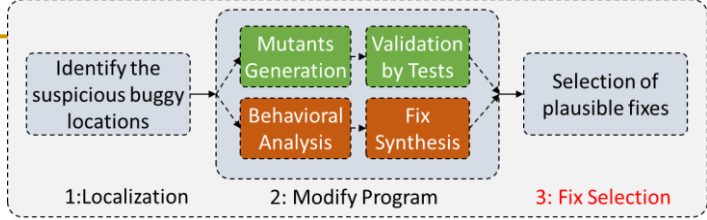
(also known as **generate-and-validate**)



ML-driven

Train a language model; apply the trained model to generate a fix (i.e., the next token) given the preceding computation (i.e., previous tokens)

Fix Selection

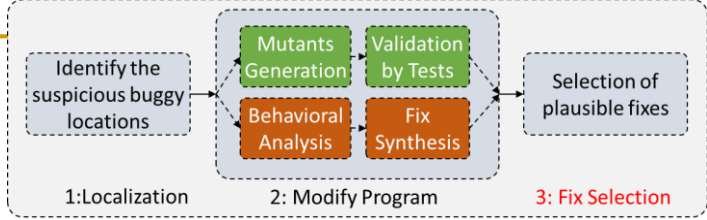


- Heuristics
  - Favors fixes that manipulate conditions over statement modification (e.g., SPR [Long et al., ESEC/FSE15])
  - Code coverage of passing tests is likely similar before and after applying a correct fix
    - Insight: A correct fix should have minimal impact on passing tests. However, an incorrect plausible fix can affect executions of passing tests, but the test oracles are too weak to reject it
    - Coverage can be structural coverage or dataflow coverage

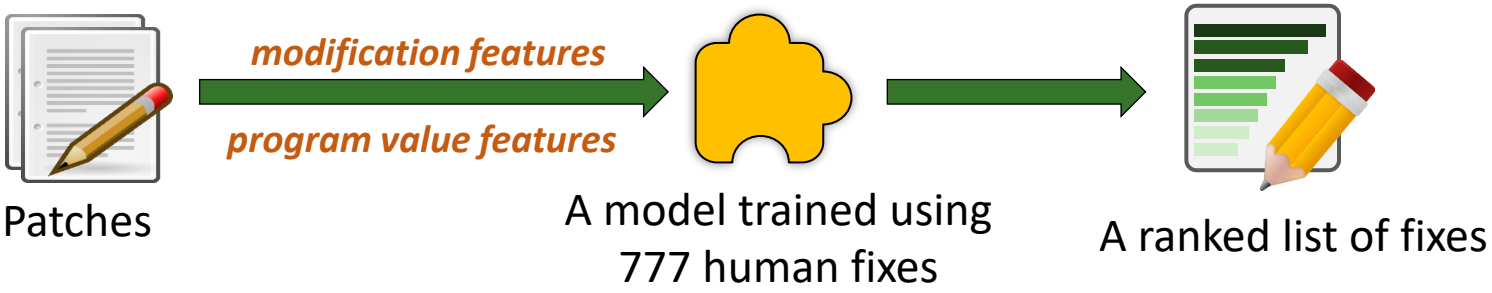
refines

  - If a fix is correct, the state of a passing test after executing the suspicious statement in the buggy version should be similar to that after executing the fixed statement in the repair version

Fix Selection



- Machine learning
  - Prophet [Long et al., POPL16] learns universal features (properties) common to known correct fixes
  - Correct fixes are likely to follow the style of human fixes and comprehensible to programmers



Comparison (Mutation-driven vs Semantics-Driven)

- *Targeting Bugs (what types of bugs can be fixed)*
- *Efficiency (the capability of fixing bugs within affordable time)*
- *Scalability (the size of the programs can be fixed)*

	Targeting Bugs	*Efficiency*	*Scalability*
Mutation-driven APR	*Generalizable to fix all types of bugs.* *e.g., null pointer exception, segmentation fault, overflow ...*	*Greatly compromised by the search-space explosion problem.*	*Simple, intuitive, and scalable to fix large programs.*
Semantics-driven APR	*Less generalizable limited by constraint solving and program synthesis.* *e.g., assignments, if-conditions ...*	*More efficient with a more tractable search space by using program synthesis.*	*Less scalable due to the overhead of symbolic execution and constraint solving.*

Empirical Study

A Recent Empirical Study [Motwani, EmSE18]

- RQ1: Can APR techniques repair important bugs?
- RQ2: Can APR techniques repair complex bugs?
- RQ3: Can APR techniques repair bugs with weak tests?
- RQ4: Are certain types of bugs difficult to repair automatically?

How to design the experiments?

- Subject preparation
- Measurement of bug importance
- Measurement of bug complexity
- Measurement of test suite quality

Appreciate the skill of acquiring the ground truth for subjective concepts

Experiment Setup

- Build two bugs benchmarks
  - Collection of 185 C (ManyBugs dataset) and 357 Java (Defects4J dataset) "fixed" bugs in real projects
  - Concern 9 C and 5 Java projects
  - Programs concerned range from 22K to 2.8M LOC
- Annotate 114 C and 205 Java bugs with
  - Bug importance
  - Patch complexity
  - Test suite quality
  - https://github.com/LASER-UMASS/AutomatedRepairApplicabilityData/

ManyBugs			
program	kLoC	defects	program description
fbc	97	3	legacy language compiler
gmp	145	2	multi-precision math library
gzip	491	5	data compression utility
libtiff	77	24	image processing library
lighttpd	62	9	web server
php	1,099	104	web programming language
python	407	15	general-purpose language
valgrind	793	15	dynamic debugging tool
wireshark	2,814	8	network packet analyzer
total	**5,985**	**185**	

Defects4J			
JFreeChart	96	26	chart drawing library
~~Closure~~	~~90~~	~~133~~	~~compiler~~
Commons Lang	22	65	Apache core library
Commons Math	85	106	Apache math and stat library
Joda-Time	28	27	date and time library
total	**321**	**357**	

Experiment Setup

- Build two bugs benchmarks
  - Collection of 185 C (ManyBugs dataset) and 357 Java (Defects4J dataset) “fixed” bugs in real projects
  - Concern 9 C and 5 Java projects
  - Programs concerned range from 22K to 2.8M LOC
- Annotate 114 C and 205 Java bugs with
  - Bug importance
  - Patch complexity
  - Test suite quality
  - https://github.com/LASER-UMASS/AutomatedRepairApplicabilityData/

Annotation – Bug Importance

- Bug priority/severity
  - Five priority levels: 1 (lowest) to 5 (highest)
- Number of project versions affected
  - Affects more versions: bug is less important
- Time taken to fix the bug
  - Time between filing the first issue report and the last commit for the issue
  - Longer fix time: bug is less important

Annotation – Bug Complexity

- Number of source files modified by the fix
  - More files: higher complexity
- non-comment LOC of the fix
  - More LOC: higher complexity

Annotation – Test Suite Quality

- Fraction of the lines modified by the fix covered by the test suite
  - Higher coverage: higher quality
- Number of bug revealing tests
  - Higher number: higher quality
- Number of tests that execute at least one line of the fix
  - Higher number: higher quality

Annotation – Patch Characteristics

- Changes data structures or types
- Changes method signatures
- Changes function call arguments
- Changes conditionals
- Adds new variables / if-statements / loops / function calls / functions

Experimental Setup (cont.)

- 9 subjects implementing 7 APR techniques
- 6 APR tools for C
  - GenProg [LeGoues et al. ICSE12], RSRepair [Qi et al., ICSE14], AE [Weimer et al. ASE13], Kali [Qi et al., ISSTA15], SPR [Long et al., ESEC/FSE15], Prophet [Long et al., POPL16]
- 3 APR tools for Java
  - Nopol [DeMarco et al., TSE16], GenProg [Martinez et al., EmSE17], Kali [Martinez et al., EmSE17]

Results

At least one of the APR tools could generate a fix

technique	patched	unpatched	patched %
ManyBugs (185-defect full set)			
GenProgC	87	98	47%
RSRepair	97	88	52%
AE	86	99	46%
∪C	**104**	**81**	**56%**
ManyBugs (105-defect subset)			
KaliC	27	78	26%
SPR	46	59	44%
Prophet	43	62	41%
∪C	**52**	**53**	**49%**
Defects4J (224-defect set)			
GenProgJ	27	197	12%
KaliJ	22	202	10%
Nopol	35	189	16%
∪Java	**47**	**177**	**21%**

RQ1: Able to fix important bugs?

Findings:

- Java-based APR techniques are moderately more likely to fix bugs of a higher priority
- Little correlation with bug fix time and number of affected versions
- Overall, bug fixing ability has little correlation with bug importance

RQ2: Able to fix complex bugs?

Findings:

- C-based APR techniques are less able to generate fixes that involve multiple lines or files
- Overall, the bug fixing ability does not have strong correlation with complex bugs
  - Instructor: Our experience is that existing APR techniques can generate mostly simple fixes (~20% of the bugs documented in Defects4J) with small search space

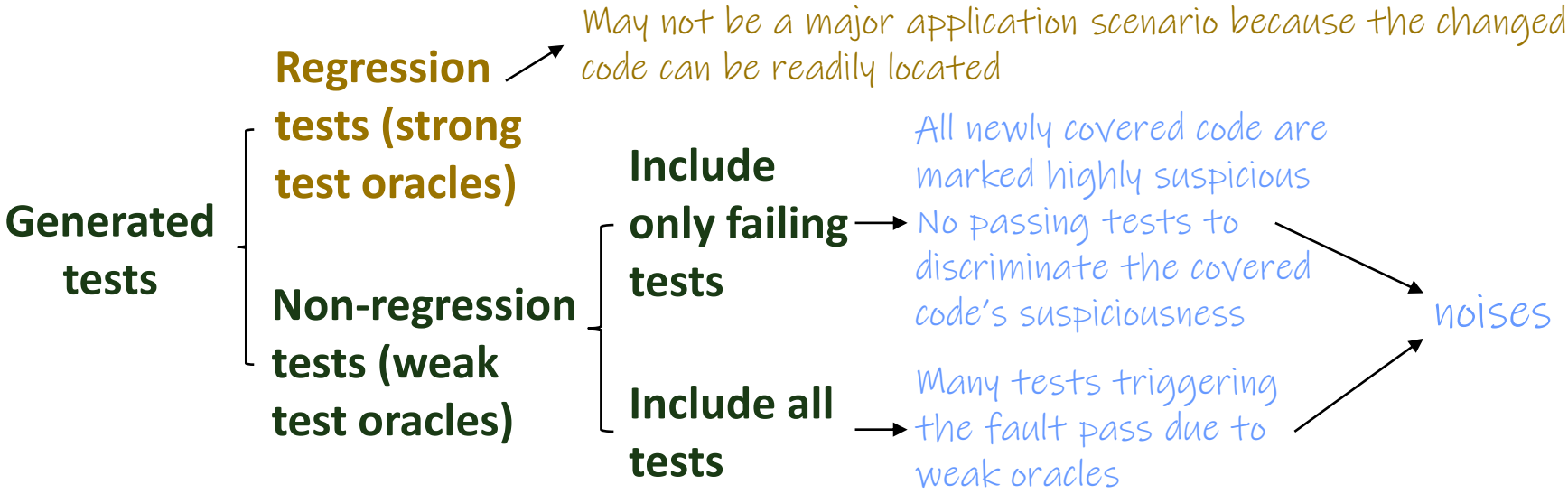
RQ3: Able to fix with weak tests?

Findings:

- Insignificant correlation with test coverage
- This means adding more failing tests and increasing test coverage are not useful
  - Instructor: Increasing test coverage alone without strengthening test oracles cannot improve test effectiveness
  - Instructor: A recent finding reveals that 5% of flaky tests can significantly reduce the accuracy of fault localization
- Less able to fix bugs detected by more failing tests
  - Instructor: The use of more generated tests does not necessarily improve APR. Why?

RQ3 (continue)

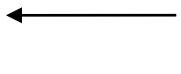
- The use of more generated tests does not necessarily improve APR. Why?



RQ4: Correlation with bug characteristics?

Findings:

- Less able to generate fixes that
  - insert new loops or function calls
  - change a method signature



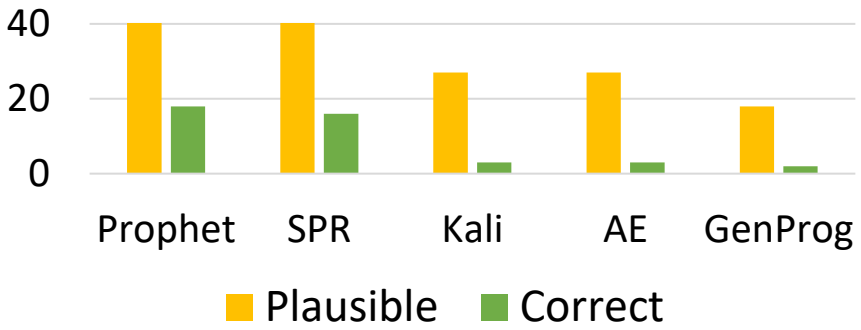
Harder problem

Can be improved with introduction of related mutation operators

Findings in Other Empirical Studies

Plausibility and Correctness [Qi et al., ISSTA15]

Plausible: Pass all the test cases and produce correct output.
Correct: Equivalent to the actual fixes.



Automated generated fixes are controversial to be deployed directly in practice.

Use case: Automated Generated Patches → Facilitating Developers Fix Bugs →

Patches as debugging aid [Tao et al., FSE14]

Industry Adoption

Industry Adoption

- Github
  - Repairnator [arXiv: 1810.05806; Urli et al., ICSE18] – Github project
  - https://medium.com/@martin.monperrus/human-competitive-fixes-in-automatic-program-repair-with-repairnator-359042e00f6a
- Facebook
  - SapFix [Marginean et al., ICSE19]
- Fujitsu
  - Elixir [Saha et al., ASE17]

Software

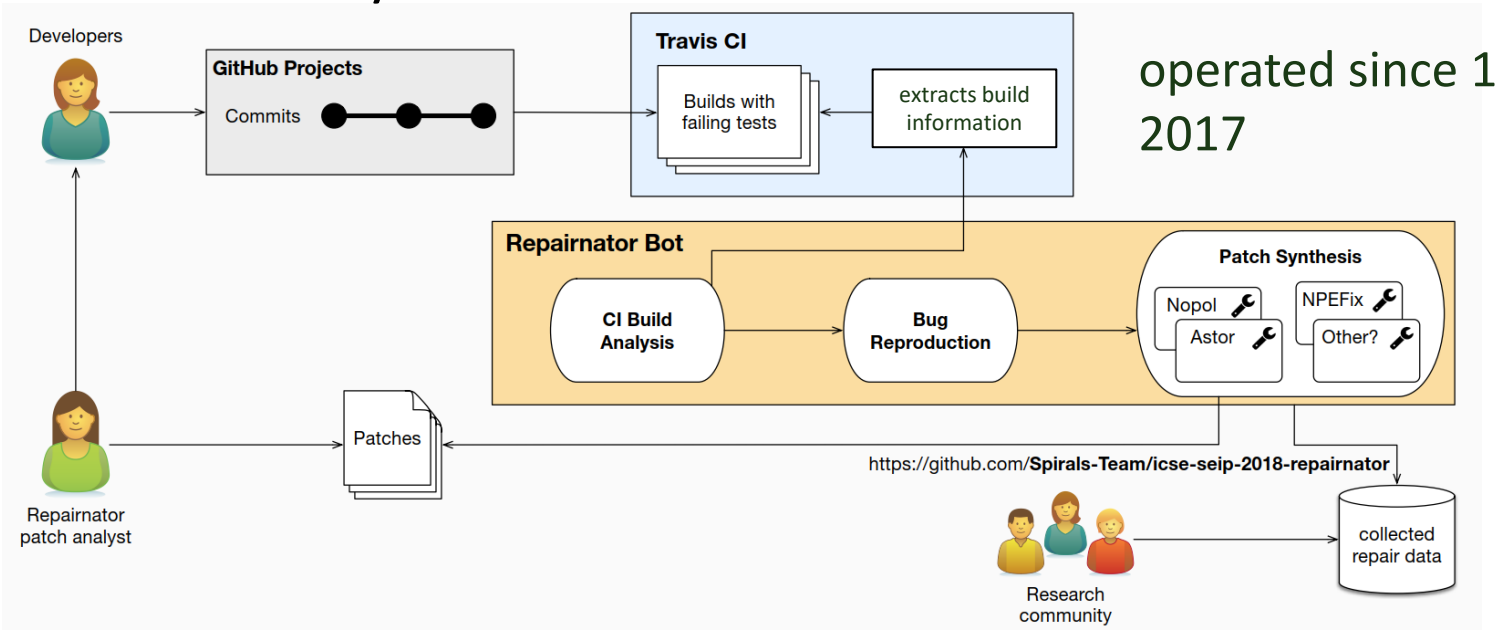
Sophie Timsit - 4/09/2018

Repairnator, an autonomous robot to repair computer bugs

Like Terminator, Repairnator is a robot equipped with an artificial intelligence. The most recent creation of the Spirals* project team, it has been developed within the framework of the technology development initiative ADT LibRepair won in 2016 by post-doctoral researcher Simon Urli. Repairnator is the first repair bot capable of automatically repairing computer bugs on a large scale. In short, effective and autonomous software that saves other software programs from code errors – and an open source solution available to all developers.

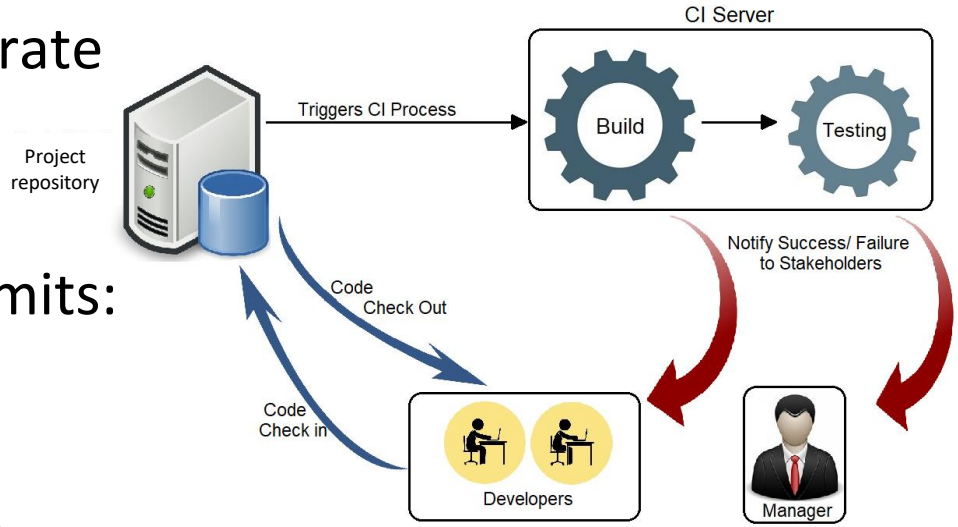
Repairnator [Urli et al., ICSE-SEIP19; Monperrus et al., arXiv:1810.05806]

- A bot automatically repairs builds of Java programs with test failures found by Travis CI on Github



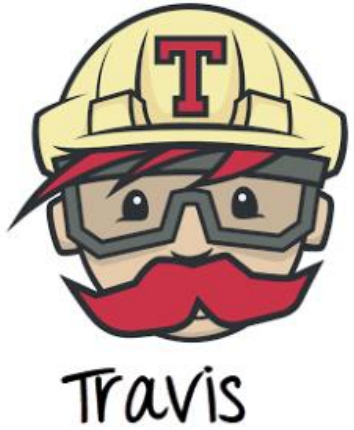
Continuous Integration (CI)

- Developers commit and integrate work continuously to project repository
- CI server integrates new commits:
  - builds system
  - conducts tests
  - assigns a build tag to the version
  - informs developers and manager about the build result (succeeds/fails)
  - Developers fix the bug if the build fails



Travis CI – Enabling Methodology for DevOps

- Travis is a popular continuous integration service on Github and Bitbucket
- Automatically generates a build and run the specified unit tests upon each code commit
- A build fails when it is not compilable or some tests fail
- 2,477 (17.46%) Java projects on Github actively use Travis CI



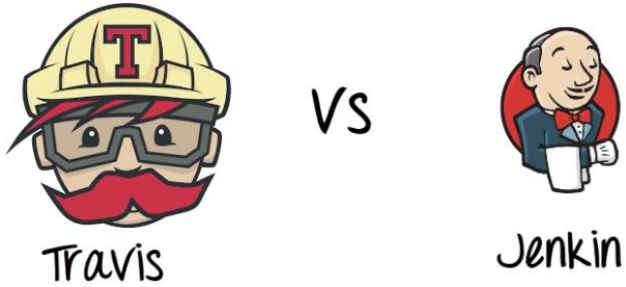
Industry Adoption of Continuous Integration

	Google DS	Rails DS
# of total commits	4,421	2,804
# of failing commits	1,022	574
Avg commit duration (sec)	948	1,505
# of distinct test suites	5,536	2,072
# of distinct failing test suites	154	203
Avg # of test suites per commit	331	1,280
# of total test suite executions	1,461,303	3,588,324
# of failing test suite executions	4,926	2,259
Test suite execution time (sec)	4,192,794	4,220,482
	1,164 cpu-hrs	1,172 cpu-hrs

- Google DS contains info based on a sample of Google products over 15 days
- Rails DS contains info based on a popular open-source project Rails (from Mar-Aug/2016)
- Commit duration = Time taken to run test suites for a commit
- Time to validate a fix can be non-trivial
- Even more expensive if including the building time

Source: Liang et al., Redefining Prioritization: Continuous Prioritization for Continuous Integration, ICSE18.

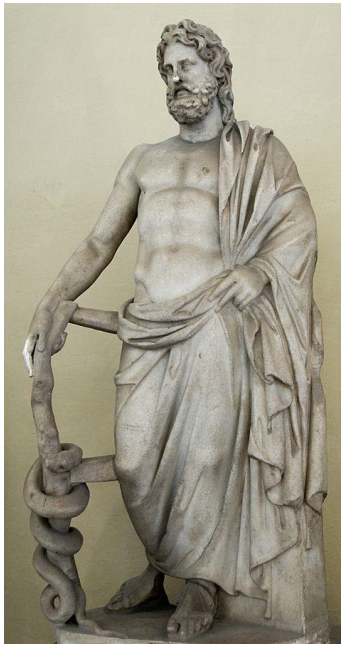
Two Popular CI Tools



Commercial project	Open-source Java project
Free for open-source projects	Free
Automatic Github integration	Requires elaborate setup and configuration
Cloud-based	Server-based
Easy to use	Highly customizable
Good for open-source projects hosted on cloud	Good for self-hosted project repository

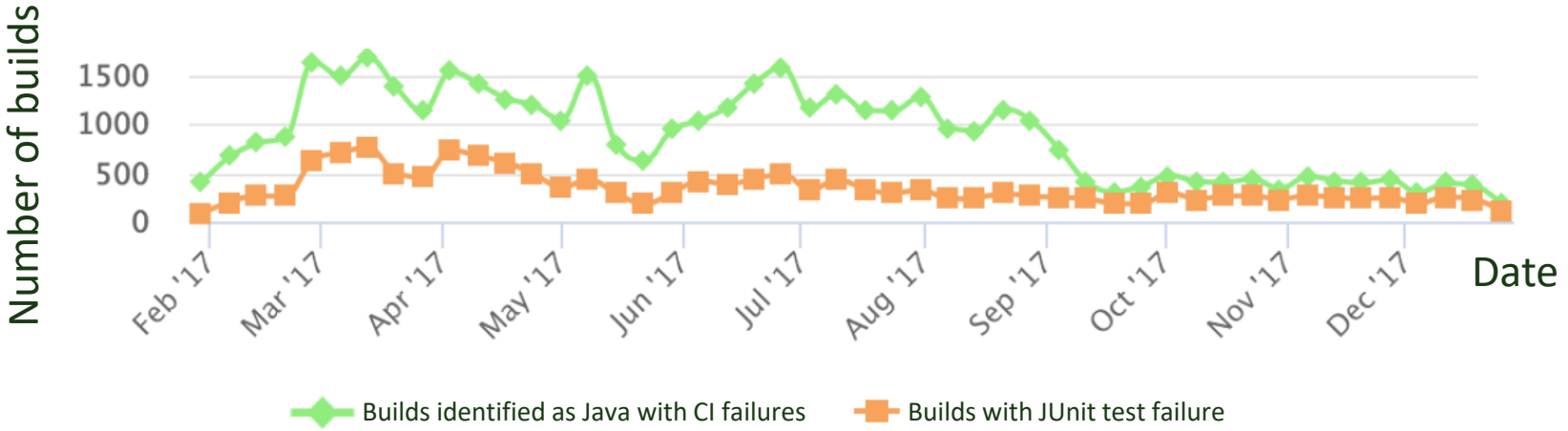
Name of the bot at Github: Luc Esape

- Looks like a software engineer
- A fake human identity of Repairnator
- Goal: avoid bias against accepting pull-requests created by a bot
- Name is an anagram of Esculape, the Roman god of ~~human~~ program healing



Luc Esape
lucesape
Engineer@Spirals-Team

Builds analyzed by Repairnator from Feb 2017 to Jan 2018



Failing Builds are Common

- Using API of Travis CI to extract build information for analysis
  - Status
  - Logs
    - Existence of test failures
  - Date
  - …

Failing builds are common

Project	Builds with tests failures	PR builds with tests failures	% of PR builds
prestodb/presto	1000	889	88.90%
druid-io/druid	579	464	80.14%
apache/flink	477	349	73.17%
apache/nifi	472	327	69.28%
hubspot/singularity	437	114	26.09%
apache/storm	349	255	73.07%
corfudb/corfudb	313	151	48.24%
spring-projects/spring-boot	277	92	33.21%
apache/zeppelin	210	47	22.38%
jabref/jabref	201	82	40.80%
Total on 468 projects	11 523	5 874	50.98%
Average on 468 projects	25	13	50.98%

Bug Reproduction

For each build failure due to failing tests, perform:

- Clone the repository
- Checkout the concerned commit
- Compile the build
- Run tests and observe outcomes

- Reproducibility varies across projects
  - About 30% of failing builds with test failure can be reproduced

Project	Builds with test failure	Reproduced bugs	% of Reproduced
druid-io/druid	579	359	62.00%
apache/flink	477	326	68.34%
prestodb/presto	1000	194	19.40%
hubspot/singularity	437	182	41.65%
corfudb/corfudb	313	126	40.26%
apache/storm	349	111	31.81%
geoserver/geoserver	118	109	92.37%
spotify/docker-client	111	99	89.19%
xetorthio/jedis	100	94	94.00%
4pr0n/ripme	94	87	92.55%
Total on 468 projects	11 523	3 551	30.82%
Average on 468 projects	25	8	30.75%

Common Failure Types

- Top 10 failure types
- Produced patches for around 80 different builds
- Top two are AssertionError and NullPointerException

Exception	Occurrences
java.lang.AssertionError	2 162
java.lang.NullPointerException	641
org.junit.ComparisonFailure	419
java.lang.Exception	250
java.lang.IllegalStateException	202
java.lang.NoClassDefFoundError	197
java.lang.RuntimeException	191
junit.framework.AssertionFailedError	163
java.lang.ExceptionInInitializerError	117
java.io.IOException	110
Subtotal	4 452
Other	1 928
Total	6 380

Overfitting Patches

- Overfitting patches are common. Examples include:
  - this.equals(this)
  - this instanceof Object

```
@@ JCommentPart.java


- if (aValue == null)
+ if (this.equals((java.lang.Object) this))
    return;
  if (aValue instanceof Object [])
```

Project	Builds w/ patches	Nopol patches	NPEFix patches
jamesagnew/hapi-fhir	1	35	0
spotify/cassandra-reaper	1	1	0
xmlunit/xmlunit	1	145	0
apache/pdfbox	1	120	0
LiveRamp/hank	1	4	0
spring-cloud/spring-cloud-dataflow	1	0	1
IQSS/dataverse	2	0	16
bonigarcia/webdrivermai	3	30	0
GeoWebCache/geowebca	1	0	2
timmolter/XChange	1	0	4
phax/jcodemodel	1	624	0
phoenixnap/springmvc-raml-plugin	1	348	0
Total	15	1 307	23

Practical Challenges (Threats to Validity)

- Spurious bugs
  - Not sure if the bugs reproduced are the ones causing build failures
- Different build scripts
  - Repairnator uses the default Maven and Gradle setup, which may differ from the one used by the original developers
- Different Travis CI environment
  - The original failing build may run at an environment different from that of Repairnator
  - Examples: a specific Java version / MacOS version / Linux version
- Existing repair tools cannot fix bugs spanning multiple modules

Outlook

- Not yet effective
- Promising
  - Increasing software development is adopting CI
  - Idea of software repair bots is feasible
- High demands from developers
- Repairnator is welcome by Github developers
- Explainable patches

Five Adopted Patches between Jan – Jun 2018

Date	Contribution	Developer comment
Jan 12, 2018	aaime/geowebcache/pull/1	"Thanks for the patch!"
Mar 23, 2018	parkito/BasicDataCodeU[...]/pull/3	"merged commit 140a3e3 into parkito:develop"
April 5, 2018	dkarv/jdcallgraph/pull/2	"Thanks!"
May 3, 2018	eclipse/ditto/pull/151	"Cool, thanks for going through the Eclipse process and for the fix."
June 25, 2018	donnelldebnam/CodeU[...]/pull/151	"Thanks!!"

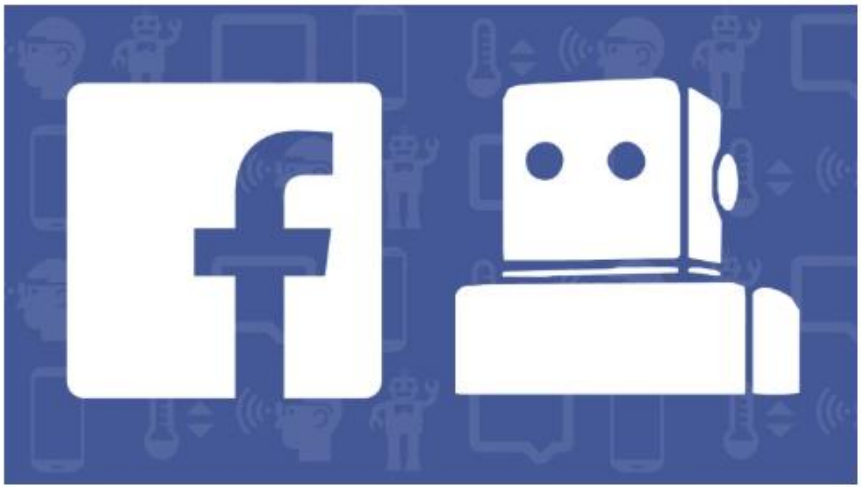
Table 1: Human-competitive contributions on Github: patches synthesized by the Repairnator robot and accepted by the human developer.

Available at: https://github.com/Spirals-Team/repairnator

Facebook's new 'SapFix' AI automatically debugs your code

Josh Constine @joshconstine / 4 months ago

Comment



13 Sep 2018

- Facebook uses artificial intelligence to automate the creation of fixes for bugs that have been identified by its software testing tool Sapienz, which is already being used in production.
- For simpler bugs, SapFix creates fixes that revert the code submission that introduced them.
- For complicated bugs, SapFix uses a collection of "templated fixes" that were created by human engineers based on previous bug fixes.

extracted from the news but it is more sophisticated

… extracted from the news

SapFix – End-to-end Repair at Facebook (May 2019)

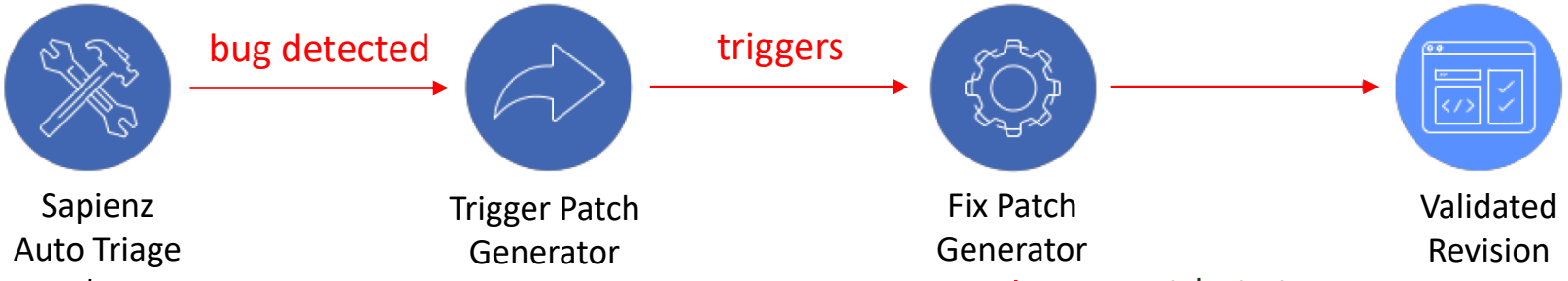
- First end-to-end fault fixing deployment
  - From test to deployed repairs
- Deployed at 6 production systems at Facebook
  - Facebook, Messenger, Instagram, FBLite, Workplace, Workchat
  - Multi-million LOC
  - Used daily by multi-million users

Source: Marginean et al., SapFix: Automated End-to-End Repair at Scale, ICSE-SEIP19

SapFix – End-to-end Repair at Facebook (May 2019)

- Automate the end-to-end repair cycle of ~75% of crashes reported by Sapienz
- To limit fix patterns, Sapfix focuses on null-dereference faults
  - Do not require additional repair ingredients
- Patch generation strategy (in decreasing order of preference)
  - Template fix
  - Mutation fix
  - Revert full diff // a commit can consist of multiple code diffs
  - Revert partial diff

Fix Generation Workflow



Sapienz selects tests for each recent code diff. Bug triage: When tests fail, identify the developer who submitted the code diff

Able to identify the diffs that possibly induce the failing tests

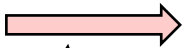
Revert subsets of diffs so that the revised program compiles successfully

Credit: Facebook

Fixing Template

- Mine (or manually designed) fix templates from previous fixes in repository
  - E.g., h0.h1(); → if (h0 == null) return; h0.h1();

```
public void onDestroyView() {
  mListView.clearListeners();
  mListView = null;
}
```

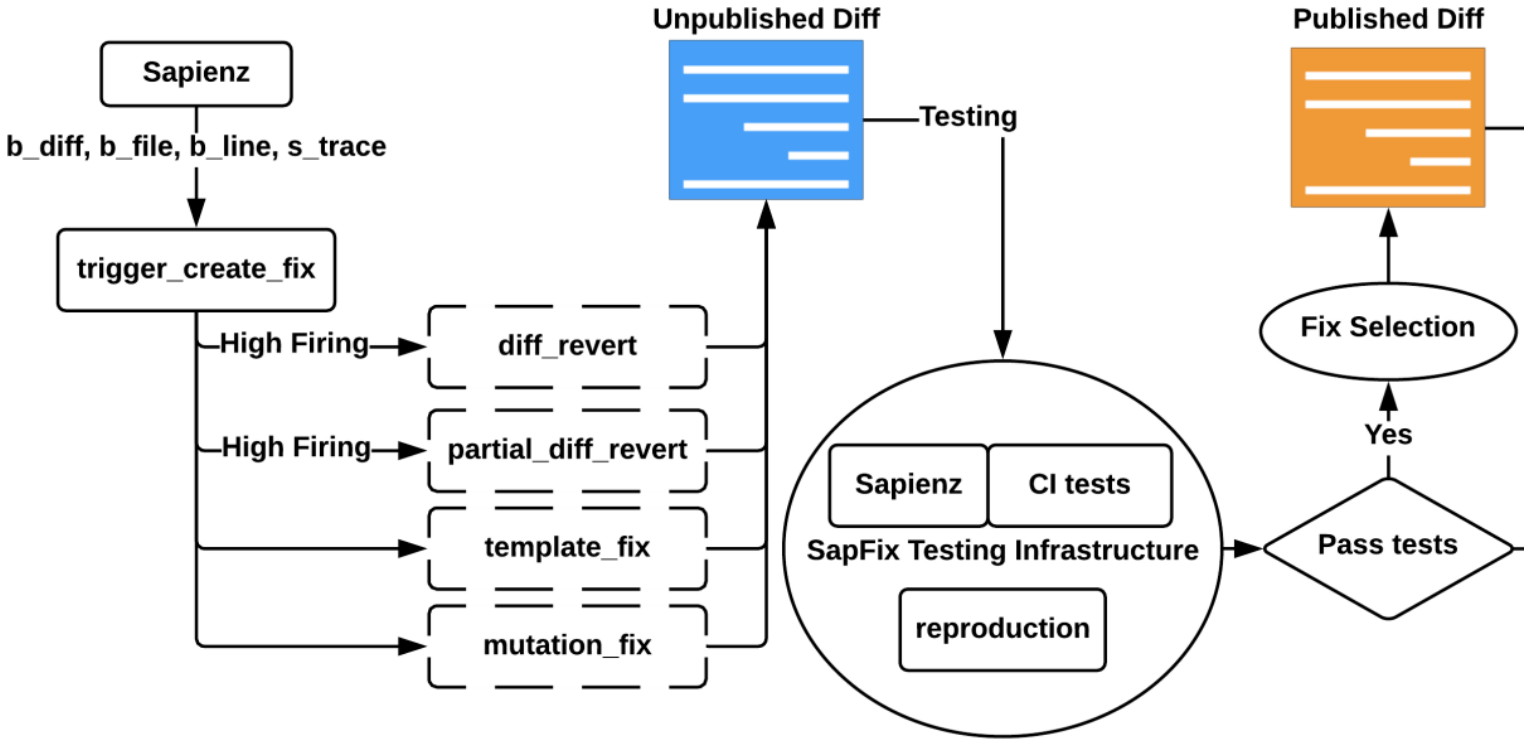


```
public void onDestroyView() {
  if (mListView == null)
    return;
  mListView.clearListeners();
  mListView = null;
}
```

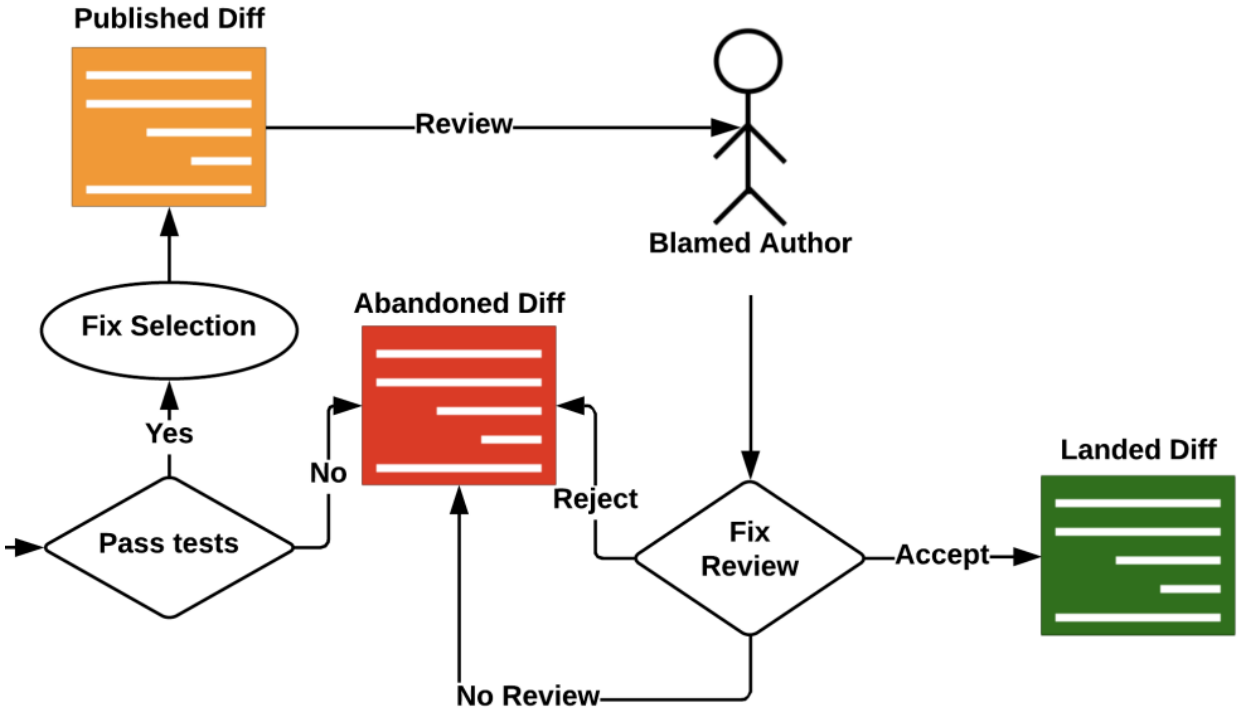
mapping

```
mListView.clearListeners()
h0          h1
```

SapFix Workflow

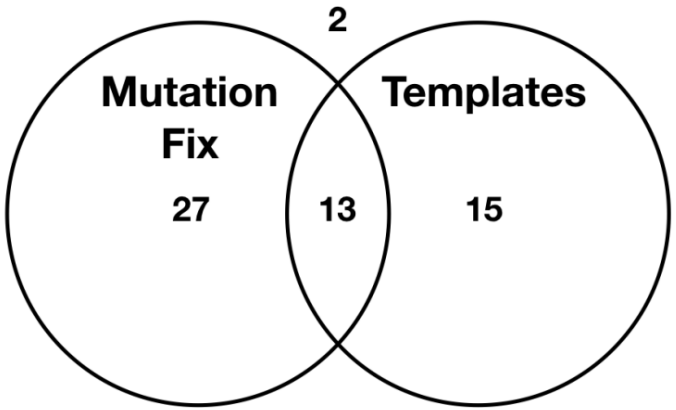


SapFix Workflow (cont.)



Experience

- Based on 57 distinct crashes reported to SapFix over three months in late 2018.
- SapFix constructed 165 patches
  - 131/165 can be correctly built and passed all tests
  - 55/131 were reported to developers covering 55/57 crashes
  - Patches were very well received
  - 40/55 can be repaired using templates
  - 28/55 can be repaired using mutation



Experience

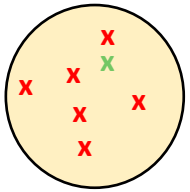
- SapFix also attempted to repair 18 crashes that could not be repaired by templates or mutation
- SapFix reverted 18 Diffs (14 fully and 4 partially)
- All Diffs were declined by developers
  - Developers were unwilling to simply revert their hard work

Conclusion

- End-to-end automated repair can work at scale in industry
- About 25% of reported repairs deployed for production
- About 25% of reported repairs were deemed correct but not landed either because the developer made minor modification of the repairs, or the developer had already fixed the bugs
- Developers showed interest in communicating with the SapFix bot
- Techniques for identifying the root causes of failures are in high demand

Challenges and Opportunities

Correctness Challenges



- Overfitting the tests: Huge amount of incorrect plausible fixes
  - Which mutation operators are likely to generate correct fixes
  - Can mutation operators be sensitive to faulty locations
- Non-functional qualities
  - Computational efficient / comprehensible / best coding practices (e.g., encapsulation, extensibility)
- Synthesize explanation of the generated fixes

Technical Challenges

- Many test suites of existing projects are inadequate
- Can test suite be strengthened upon adoption of a fix?
- Alternative validation mechanisms other than tests
  - FixMeUp [Son et al., NDSS13] validates fixes by rules
- Better search techniques
- Better and more benchmarks
  - Currently, no benchmarks on lambdas, Python and Android programs

State-of-the-art

Much room for improvement

- Four major benchmarks
  - Defect4J v1.2 (395 bugs with fixes) – 26.2% repair rate
  - Defect4J v2.0 (additional 420 bugs with fixes) – 4.5% repair rate
  - IntroClassJava (297 bugs with fixes) – 11.8% repair rate
  - QuixBugs (40 bugs with fixes) – 42.5% repair rate
- Zhu et al., Recoder, ESEC/FSE 21 *// state-of-the-art 21*
- Wong et al., VarFix, ESEC/FSE 21

Additional readings

- Online resources
  - http://program-repair.org
  - M. Monperrus. The Living Review on Automated Program Repair (March 2020). http://bit.ly/2CehUt5
  - Video: https://vimeo.com/369403234 (3:10 min)
- Survey
  - Luca Gazzola et al., Automatic Software Repair, ICSE 2018.
  - Martin Monperrus. Automatic Software Repair, ACM Computing Survey 51(1), 2018.
- A key future direction: Combine Mutation-driven, Semantics-driven and big data/AI to improve the recall while preserving high precision
  - Jiang & Xiong found that 90% of Defect4J bugs can be manually fixed by a developer without knowing the original requirements [CoRR abs/1705.04149]

Additional readings

- Claire Le Goues, Michael Pradel and Abhik Roychoudhury. Automated Program Repair. Communications of the ACM 62(12): 56-65 (2019)
- Luca Gazzola, Daniela Micucci and Leonardo Mariani. Automatic Software Repair: A Survey, TSE (2019)
- Martin Monperrus. Automatic Software Repair: A Bibliography. ACM Comput. Surv. 51(1): 17:1-17:24 (2018)
- Manish Motwani, Sandhya Sankaranarayanan, René Just, Yuriy Brun. Do automated program repair techniques repair hard and important bugs? Empirical Software Engineering 23(5): 2901-2947 (2018)
- Yuzhen Liu, Long Zhang, Zhenyu Zhang. A Survey of Test Based Automatic Program Repair. JSW 13(8): 437-452 (2018)
- Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, Martin Monperrus. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. Empirical Software Engineering 22(4): 1936-1964 (2017)

Epilogue of Automated Tools

Myths of automated tools?

Only be used on small programs?

- There are various useful automated tools developed in recent 10 years
- Many of these tools have been deployed for open-source projects and at large enterprises
- Effectiveness of tools have been demonstrated using large scale case studies

Only used for complete programs?

- **Static tools** can be applied to incomplete programs
- Does not require program execution
- Some tools even support analysis at byte code level
- Check both functional and non-functional properties
- Static tools: SpotBugs, Infer

Is random testing limited to generation of simple parameter values?

- Can generate executable code for JUnit tests!
- Automatically chain multiple method calls
- Automatically construct complex data structures.
- Deploy feedback mechanism
- Random test generation tool: Randoop

High test coverage = Adequate testing?

- Code coverage is only meaningful if it is accompanied by effective test oracles to judge correctness of test outcome
- Mutation coverage considers not only code coverage but also strength of test oracles
- Mutation coverage provides a better measurement of test adequacy
- Mutation tool: Pitclipse

Search-based test generation

- Search is driven by carefully designed fitness functions
- Deploy concolic testing (a.k.a. dynamic symbolic execution) to increase coverage
- Tool: Evosuite, Sapienz

Debugging is tedious

- Automated techniques to locate likely faulty code and repair faults passing all tests.
- Based on Reachability-Infection-Propagation model.
- Fault Localization Tools: GZoltar
- Fault Repair Tools: Repairnator and SapFix

EACH TOOL HAS ITS LIMITATIONS!

Proficient developers use automated tools wisely to facilitate their work.

Open Research Problems and Challenges

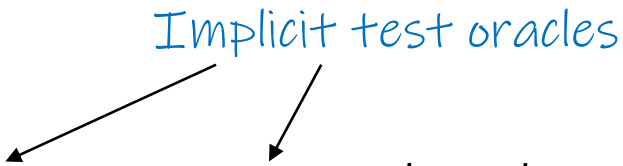
Needs from industry on bug detection tools (including both static and dynamic tools)

Open Problems

- Flaky tests
  - Failing tests may not fail reliably on every execution even when all controllable variables are held constant
  - Formulation of software testing in a probabilistic setting
- Fix detection
  - Decide whether a bug has been (correctly) fixed
  - A bug can cause a system to fail in different ways
  - Eliminating only the known failures do not necessarily fix the bug
  - How to assure that the fix does not induce a new bug
  - How to correctly cluster different failures caused by the same bug
    - Bucketing

Open Problems

- Better test oracles
  - Largely based on software crashes or memory-related
  - Requires generating more sophisticated oracles
- Fitness evaluation
  - Some fitness like fault detection rate can be expensive to evaluate
  - Search for better but computationally efficient fitness evaluation

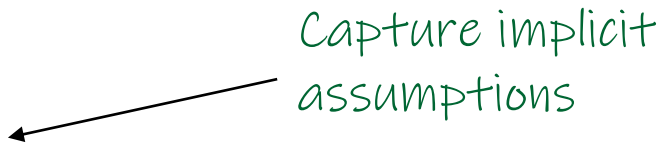


Open Problems

- Model user/hardware state and user environment
  - Generate tests that attain adequate coverage of both software and hardware state (e.g., in mobile system or IoT testing)
    - Example: The app user changes the phone orientation while the app is running
  - User state refers to a user's social state
    - Example: The user has responded to at least one post made by another user
  - User environment is agnostic to applications
    - Example: Poor GPS or Wifi signal strength; memory tight; low in battery; a phone call occurs
    - Implies wider search space

Open Problems

- Carving unit tests from system tests
  - Reduce false positive problem in unit testing
- Generating less useless mutants in automated program repair
  - Search for plausible fixes is ineffective: Only a tiny portion of the generated mutants are plausible fixes
- Combine manually-written and machine-generated tests (collaborative testing)
  - Extract test inputs and/or assertions from manually-written tests
  - Adapt these test inputs and assertions during test generation
  - Extract metamorphic relations from manually-written tests



Metamorphic Testing 蛻變測試

More Open Problems

- Automated fixing of system configuration
- Automated performance improvement
- Automated fixing of resource leakages
- Automated insertion of log statements
- Automated generation of stubs for mock objects in unit testing
- Automated test generation for DL systems, DL libraries, …
- Automated testing and debugging for cloud services, blockchain systems, smart contracts, DApps, spreadsheets, hybrid apps, …