Precise Condition Synthesis for Program Repair

Reporter: Bo Wang[1]

Authors: Yingfei Xiong[1], Jie Wang[1], Runfa Yan[2],
Jiachen Zhang[1], Shi Han[3], Gang Huang[1], Lu Zhang[1]

[1]Peking University

[2]University of Electronic Science and Technology of China

[3]Microsoft Research Asia

About me

Bo Wang (王 博)
Ph.D Student (3rd year), **Peking University**
Supervised by **Prof. Yingfei Xiong**

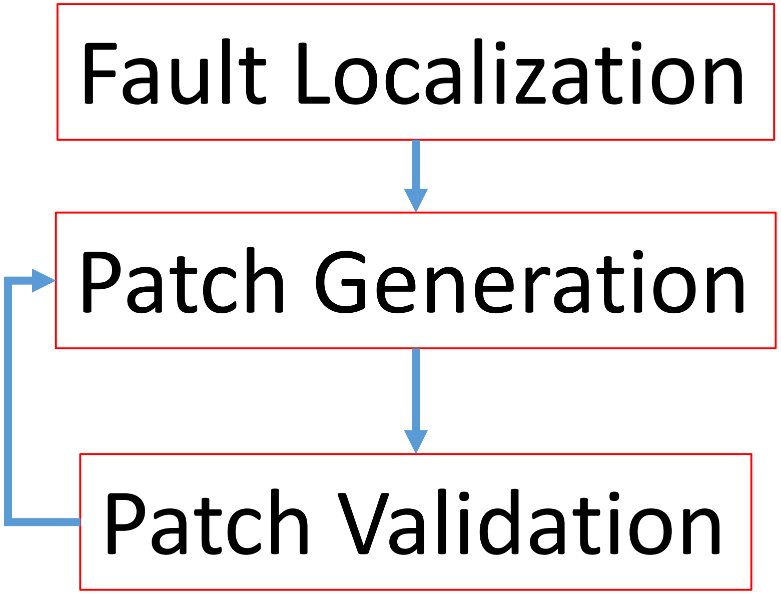
Software testing

- Faster Mutation Analysis via Equivalence Modulo States. ISSTA’17, ACM SIGSOFT Distinguished Paper Award
- Dynamic analysis of shared execution in software product line testing. Doctoral Symposium of SPLC’16

Program automated repair

Test-Based Program Repair

Input: A program and a test suite, with at least a failed test
Output: A patch that makes the program pass all tests



"Generate-Validate" Framework

GenProg, PAR, SemFix, Nopol, DirectFix, SPR, QACrashFix, Prophet, Angelix, …

Precision

- The problem of **weak test suites** [Qi-ISSTA15]
  - Test suites in real world projects are often too weak to guarantee patch correctness

- Precision = $\dfrac{\#\mathit{Correctly\ Repaired\ Defects}}{\#\mathit{All\ Defects\ with\ Patches}}$
- Precision of existing approaches[1]

jGenProg	18.5%[2]
Nopol	14.3%[2]
Prophet	38.5%[3]
Angelix	35.7%[3]

[1] If multiple patches are generated for one defect, only the fist is considered

[2] Evaluated on Defects4J benchmark

[3] Evaluated on ManyBugs benchmark

Goal of This Talk

- Goal: to repair programs with a high precision
- Targeted defect class: condition bugs

```
  lcm = Math.abs(a+b);
+ if (lcm == Integer.MIN_Value)
+   throw new ArithmeticException();
```

Missing boundary checks

```
-  if (hours <= 24)
+ if (hours < 24)
      withinOneDay=true;
```

Conditions too weak or too strong

Condition bugs are common

ACS System

- ACS = Accurate Condition Synthesis
- Two sets of templates for repair

Oracle Returning

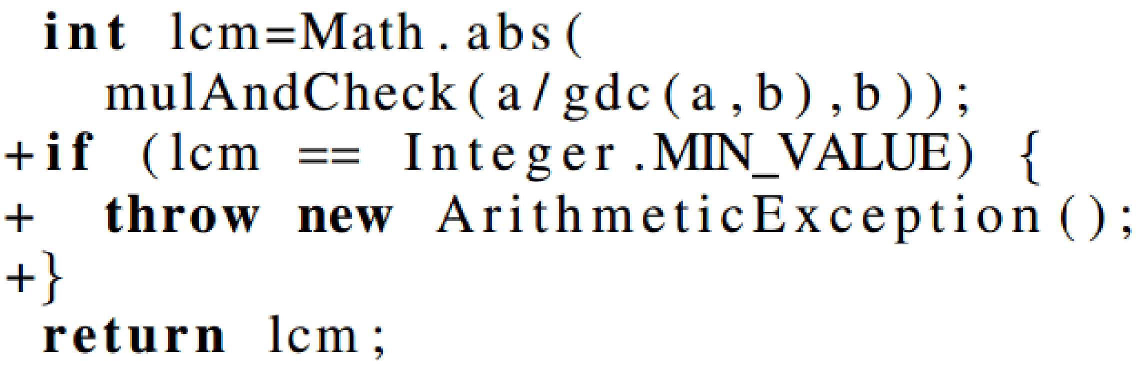
- Inserting one of the following statement before the last executed statement
  - if ($C) throw ${Expected Exception};
  - if ($C) return ${Expected Output};

Condition Modifying

- Changing the condition located by predicate switching
  - if ($D) => if ($D || $C)
  - if ($D) => if ($D && $C)

Need to synthesize condition $C

Challenge – Many incorrect conditions pass the tests



```
 int lcm=Math.abs(
   mulAndCheck(a/gdc(a,b),b));
+if (lcm == Integer.MIN_VALUE) {
+   throw new ArithmeticException();
+}
 return lcm;
```

Test 1 (Passed):
  Input: a = 1, b = 50
  Oracle: lcm = 50

Test 2 (Failed):
  Input: a = Integer.MIN_VALUE, b = 1
  Oracle: Expected(ArithmeticException)

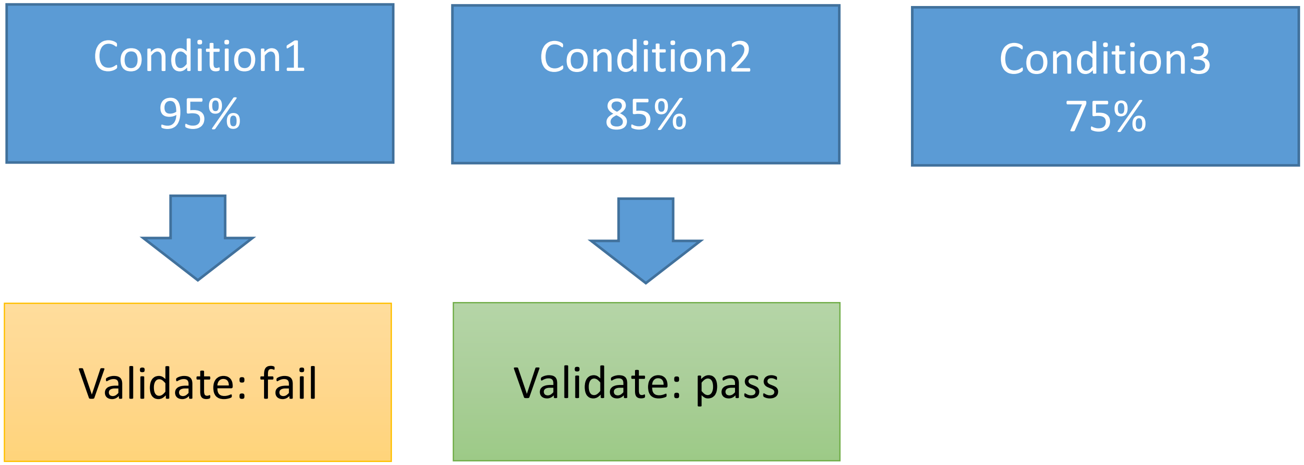
Correct condition:
lcm == Integer.MIN_VALUE

Incorrect conditions:

- a != 1
- b == 1
- lcm != 50
- …

Idea: Rank the Conditions

- Rank potential conditions by their probabilities of being correct
- Validate the conditions one by one
- Stop validating when the probability is too low



Condition1
95%

Condition2
85%

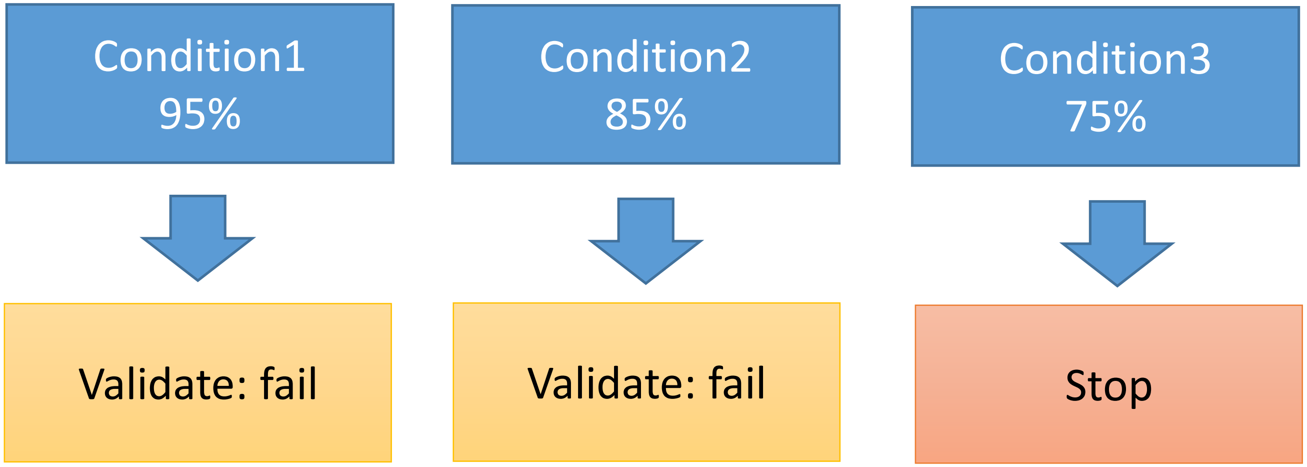
Condition3
75%

Validate: fail

Validate: pass

Idea: Rank the Conditions

- Rank potential conditions by their probabilities of being correct
- Validate the conditions one by one
- Stop validating when the probability is too low



Condition1
95%

Condition2
85%

Condition3
75%

Validate: fail

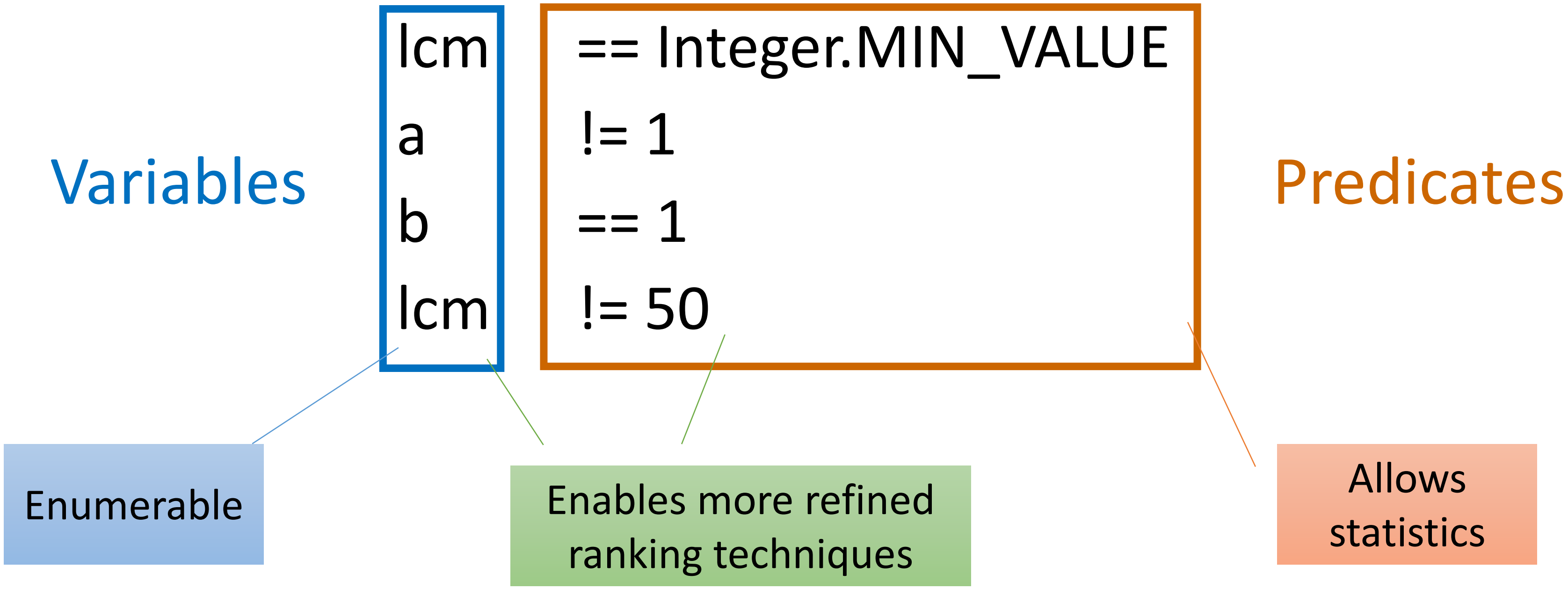
Validate: fail

Stop

Ranking Conditions is Difficult

- The number of potential conditions is large
  - Cannot enumerate the conditions
  - Difficult to perform statistics
- Why cannot simply utilize statistic?
  - Numerous conditions in the real world projects
  - The probability of a single condition is extremely low. Such as, *len < 1 , length < 1, arrLen < 1…*

Solution: Divide-and-Conquer



Variables

lcm == Integer.MIN_VALUE

a != 1

b == 1

lcm != 50

Predicates

Enumerable

Enables more refined ranking techniques

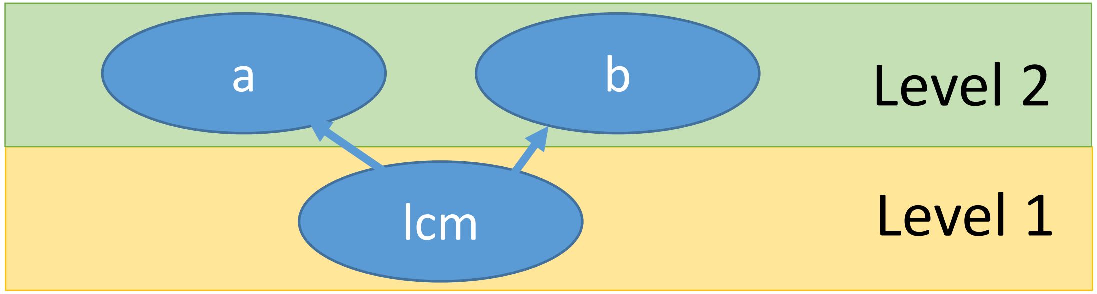
Allows statistics

Step 1: Rank variables

Step 2: Rank predicates for each variable

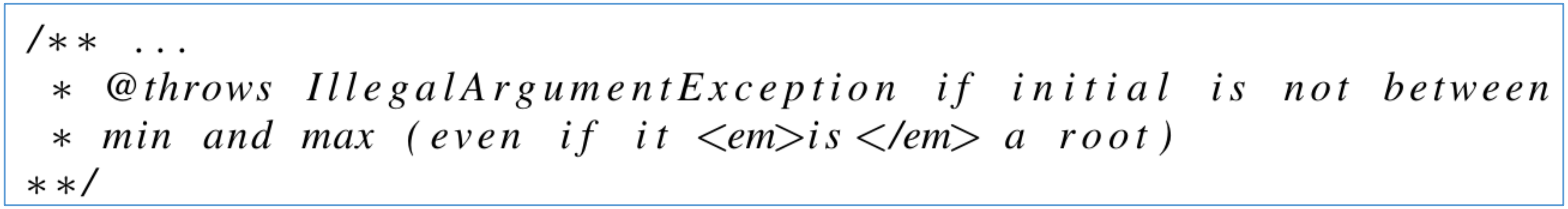
Ranking Method 1: Rank Variables by Data-Dependency

- **Locality of variable uses**: recently assigned variables are more likely to be used
- Rank variables by data-dependency
  - lcm = Math.abs(mulAndCheck(a/gdc(a, b), b))



- Consider only variables in the first two levels

Ranking Method 2: Filter Variables by JavaDoc



```
/** ...
 * @throws IllegalArgumentException if initial is not between
 * min and max (even if it <em>is</em> a root)
**/
```

Only variable “initial” is considered when throwing IllegalArgumentException

Ranking Method 3:
Rank Predicates by Context

- The predicates tested on the variables are related to its context

Variable Type

```
Vector v = …;
if (v == null) return 0;
```

Variable Name

```
int hours = …;
if (hours < 24)
      withinOneDay=true;
```

Method Name

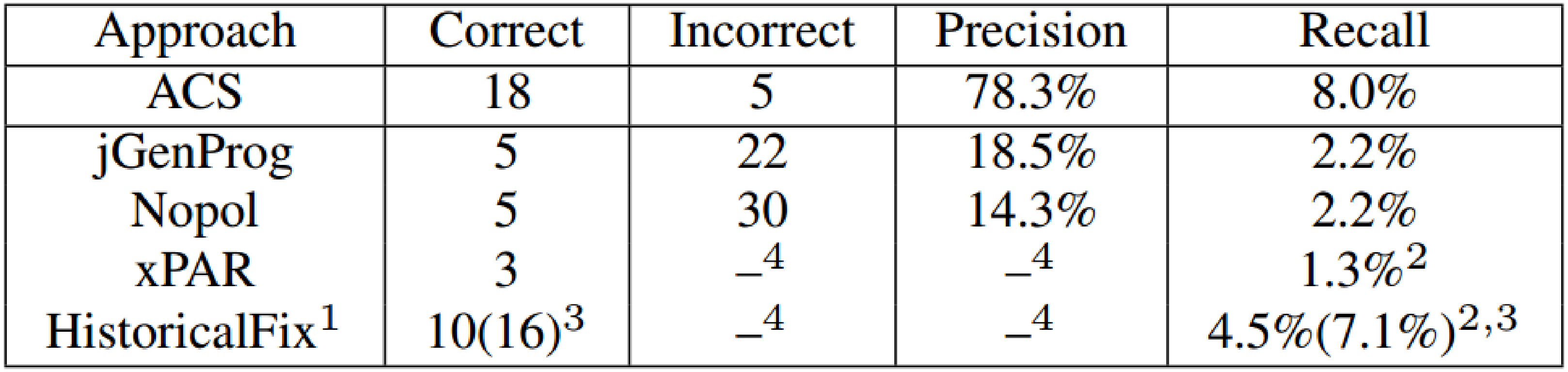
```
int factorial() {
   …
   if (n < 21) {
      …
```

- Approximate the conditional probabilities by querying GitHub
- Consider only the predicates whose probabilities are larger than a threshold

Evaluation: Performance of ACS

Dataset: Four Java projects from Defects4J benchmark:

- Apache-Time, Apache-Lang, Apache-common-Math, Jfree-Chart
- In total 224 defects



Approach	Correct	Incorrect	Precision	Recall
ACS	18	5	78.3%	8.0%
jGenProg	5	22	18.5%	2.2%
Nopol	5	30	14.3%	2.2%
xPAR	3	–[4]	–[4]	1.3%[2]
HistoricalFix[1]	10(16)[3]	–[4]	–[4]	4.5%(7.1%)[2,3]

Conclusion

- Can programs be automatically repaired with a high precision?
    - Yes, at least as high as 78.3%
- How can programs be repaired with a high precision?
    - Rank the patches by their probabilities of correctness
    - Stop when the probability is too low
- How can we rank them?
    - Divide-and-conquer with refined ranking techniques

Thank you！