Precise Condition Synthesis for Program Repair

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
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

Fault Localization

Patch Generation

Patch Validation

“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

\[
\text{Precision} = \frac{\#\text{Correctly Repaired Defects}}{\#\text{All Defects with Patches}}
\]

• 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 first 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();
```

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

Missing boundary checks

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

```java
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
Ranking Conditions is Difficult

• The number of potential conditions is large
  • Cannot enumerate the conditions
  • Difficult to perform statistics: not enough samples for each condition
Solution: Divide-and-Conquer

Step 1: Rank variables
Step 2: Rank predicates for each variable

Variables
- lcm
- a
- b
- lcm

Predicates
- \( \text{lcm} \) == Integer.MIN_VALUE
- \( a \) != 1
- \( b \) == 1
- \( b \) != 50

Enumerable

Enables more refined ranking techniques

Allows statistics
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
  • \(\text{lcm} = \text{Math.abs(mulAndCheck}(a/\text{gdc}(a, b), b))\)

• Consider only variables in the first two levels
Ranking Method 2: Filter Variables by JavaDoc

```java
/** ...
 * @throws IllegalArgumentException if initial is not between min and max (even if it is 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

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Variable Name</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>v = ...;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (v == null) return 0;</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>hours = ...;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (hours &lt; 24)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>withinOneDay=true;</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>factorial() {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (n &lt; 21) {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- Approximate the conditional probabilities by querying GitHub
- Consider only the predicates whose probabilities are larger than a threshold
Evaluation: Performance of ACS

Dataset: Four projects from Defects4J benchmark:
• Time, Lang, Math, Chart
• In total 224 defects

<table>
<thead>
<tr>
<th>Approach</th>
<th>Correct</th>
<th>Incorrect</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACS</td>
<td>18</td>
<td>5</td>
<td>78.3%</td>
<td>8.0%</td>
</tr>
<tr>
<td>jGenProg</td>
<td>5</td>
<td>22</td>
<td>18.5%</td>
<td>2.2%</td>
</tr>
<tr>
<td>Nopol</td>
<td>5</td>
<td>30</td>
<td>14.3%</td>
<td>2.2%</td>
</tr>
<tr>
<td>xPAR</td>
<td>3</td>
<td>4</td>
<td>_4</td>
<td>1.3%</td>
</tr>
<tr>
<td>HistoricalFix¹</td>
<td>10(16)³</td>
<td>_4</td>
<td>_4</td>
<td>4.5%(7.1%)²,³</td>
</tr>
</tbody>
</table>
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!