Precise Program Repair

Yingfei Xiong
Assistant Professor under Young Talents Plan
Peking University
China
Background of Yingfei Xiong

• 2006~2009, The University of Tokyo, Ph.D.
  • Advisor: Zhenjiang Hu, Masato Takeichi
• 2009~2011, University of Waterloo, PostDoc
  • Supervisor: Krzysztof Czarnecki
• 2012~, Peking University, Assistant Professor under Young Talents Plan
• Research Interests
  • Software Analysis, Program Language Design
Origin

• “War. War never changes” – Fallout series
  • The war between developers and bugs never changes

• Fault Detection: Is there a bug?
  • Since 60s
  • Example Techniques: Testing, Verification

• Fault Localization: Where is the bug?
  • Since 90s
  • Example Techniques: Spectrum-based fault localization

• Fault Repair: How to fix the program?
  • Since 00s
  • Example Techniques: Test-based Program Repair
“Generate-Validate” Program Repair

Input: a program and a set of tests, where the program fails at least one test
Output: a patch that makes the program pass all the tests

- Fault Localization
- Patch Generation
- Patch Validation
Existing Work

• GenProg
  • [Weimer et al.: ICSE’09, GECCO’09, CACM’10, ICSE’12]
  • Approach
    • Replace the potentially faulty code with code pieces elsewhere
    • Use search algorithm to find an optimal combination
  • Results: 55/105, 8$/bug

• Inspired a wave of program repair research
  • AutoFix, Nopol, RSRepair, MintHint, AutoRepair, SemFix, DirectFix, SPR...
A Turning Point

- [Qi-ISSTA’15]
  - Only 2 among the 55 defects were correctly fixed by GenProg
  - Reason: passing the test does not guarantee correctness

- [Le Goues-FSE’15]
  - Extensive experiments on more methods, datasets, test suites
  - The finding still holds

- Other work
  - Prophet, Angelix
  - The precision (proportion of correct patches) is lower than 40%
Our Work

High Precision Defect Repair

Learn from QA sites [ASE15]

Precise Condition Repair [ICSE17]

Filtering out wrong patches [ICSE18]

[ASE15] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, Hong Mei. Fixing Recurring Crash Bugs via Analyzing Q&A Sites. ASE'15

[ICSE17] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, Lu Zhang. Precise Condition Synthesis for Program Repair. ICSE'17

[ICSE18] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, Gang Huang, Identifying Patch Correctness in Test-Based Automatic Program Repair. ICSE’18
Fixing from QA Sites

• How do developers get their experience?

```java
29     public void onReceive (final Context context, final Intent intent) {
30         final int action = intent.getExtras().getInt(KEY_ACTION, -1);
31         final float bl = BatteryHelper.level(context);
32         LOG.i("AlarmReceiver invoked: action=%s bl=%s.", action, bl);
33         switch (action) {
34             ... ...
35         }
36     }
```

java.lang.RuntimeException: Unable to start receiver
com.vaguehope.onosendai.update.AlarmReceiver:
Fixing from QA Sites

Challenge: Understanding natural language is hard
Learn from QA Sites

• Observation: programmers communicate in programming languages

• Solution: Directly compare the code pieces

When I reboot my phone then also it showing same error... otherwise it is working fine.
Approach Overview

1. Crash trace

2. Q&A pages
   - Q
   - A
   - Q
   - A

3. Patch candidates
   - Exception:
     - at:
   - code

4. Fixed source code

Edit Scripts
Experiments

• 24 Android crash bugs that have answers on StackOverflow
  • Selected out of 161 Android crash bugs

• Correctly Fixed： 8
• Wrongly Fixed： 2
• Precision： 80%
• Recall： 33% (5% among Android crash bugs)
Can we repair more bugs?
Precise Condition Synthesis

- Targeted defect class: condition bugs

```java
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

Condition 1: 95%
Condition 2: 85%
Condition 3: 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% Validate: fail
- Condition2 85% Validate: fail
- Condition3 75% Stop
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

Variables:
- lcm
- a
- b
- lcm

Predicates:
- lcm == Integer.MIN_VALUE
- a != 1
- b == 1
- lcm != 50

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

Enumerable:
- Enables more refined ranking techniques

Allows statistics:
- 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
  - \( \text{lcm} = \text{Math.abs} (\text{mulAndCheck}(a / \text{gdc}(a, b), b)) \)
- 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>factorial()</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>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>17</td>
<td>6</td>
<td>73.9%</td>
<td>7.5%</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>
Can we further improve precision?

Analysis: Weak test suites are the reason for low precision
Idea: Can we improve test suites?
Improve Test Suites

- Test
- Test Input (Can be generated)
- Test Oracle (Difficult to generate)
PATCHSIM - Heuristic for test oracles

Passing tests
- Behavior on unpatched program
- Behavior on patched program
  - Similar

Failing tests
- Behavior on unpatched program
- Behavior on patched program
  - Different
TESTSIM – Heuristic for test inputs

Behavior of the new test Similar Behavior of a passing test

The new test probably passes

Behavior of the new test Similar Behavior of a failing test

The new test probably fails
Experiment Results

• 139 Patches generated by different tools
  • 30 correct, 109 incorrect

• Successfully filtered out 56.9% wrong patches, filtering out no correct patch
• Improved the precision of ACS to 85%
Vision

• Long-term Goal: automate programming
• Roadmap: deal with more and more difficult issues
  • Issue = bug report + feature requests
Conclusion

• Will program repair be useful in practice?
  • Increasing precision is the key

• Can we improve precision?
  • Yes, at least for incorrect conditions and crashes

• How can we improve precision?
  • By learning from existing resources