Design of Repair Operators for Automated Program Repair

Shin Hwei Tan
National University of Singapore
What is automated program repair?

Given a failing Test $T$, buggy program $P$

1. Fault localization – Where to fix?
2. Patch Generation using repair operators – How to fix?
3. Patch Validation – Are all tests passing?
How to extract useful repair operators?

<table>
<thead>
<tr>
<th></th>
<th>GenProg [ICSE '12]</th>
<th>relifix [ICSE '15]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Search</strong></td>
<td>• Genetic Programming</td>
<td>• Random Local Search</td>
</tr>
<tr>
<td><strong>Operators</strong></td>
<td>Mutations &amp; crossovers</td>
<td>Contextual Operators</td>
</tr>
<tr>
<td><strong>Extracted from</strong></td>
<td>Genetic Operators</td>
<td>Human Repair of Software Regression &amp; investigation of types of regressions</td>
</tr>
</tbody>
</table>
How to repair?

Regression!

Regression Fixed!
Types of Software regressions

- Changes break existing functionality
  Repair: Roll back to previous version

- Changes unmasks existing bug
  Repair: Re-mask problematic change

- Changes introduce bug in other unchanged parts
  Repair: Re-mask problematic change

- Changes unmask problematic change
  Repair: Re-mask problematic change

formulate the software regression repair problem as problem of reconciling problematic changes
## Most frequently used Operators in Human Repair

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add condition</td>
<td>Non-contextual</td>
<td>27</td>
</tr>
<tr>
<td>Add statement</td>
<td>Non-contextual</td>
<td>21</td>
</tr>
<tr>
<td>Use changed expression as input for other operator</td>
<td>Contextual</td>
<td>13</td>
</tr>
<tr>
<td>Revert to previous statement</td>
<td>Contextual</td>
<td>11</td>
</tr>
<tr>
<td>Replace with new expression</td>
<td>Non-contextual</td>
<td>13</td>
</tr>
<tr>
<td>Remove incorrectly added statement</td>
<td>Contextual</td>
<td>9</td>
</tr>
<tr>
<td>Change type</td>
<td>Non-contextual</td>
<td>5</td>
</tr>
<tr>
<td>Add method</td>
<td>Non-contextual</td>
<td>5</td>
</tr>
<tr>
<td>Add parameter</td>
<td>Non-contextual</td>
<td>4</td>
</tr>
<tr>
<td>Add local variable</td>
<td>Non-contextual</td>
<td>3</td>
</tr>
<tr>
<td>Swap changed statement with neighbouring statement</td>
<td>Contextual</td>
<td>2</td>
</tr>
<tr>
<td>Negate added condition</td>
<td>Contextual</td>
<td>1</td>
</tr>
<tr>
<td>Convert statement to condition variable statement</td>
<td>Contextual</td>
<td>1</td>
</tr>
<tr>
<td>Add field</td>
<td>Non-contextual</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>116</strong></td>
</tr>
</tbody>
</table>
Contextual Operators

- Use changed expression as input for other operator

- if (((f = lookup_file (p)) != 0 && f->is_target)
+ if (((f = lookup_file (p)) != 0 && (f->is_target || intermed_ok))

- Revert to previous statement

- /* Removing this loop will fix Savannah bug #16670: 
- do we want to? */
- while ( out > line && isblank (( unsigned char ) out[-1]))
- --out ;
Experimental Results

- Evaluated on 7 open source projects
  - *relifix* repairs 23 bugs, *GenProg* only fixes five bugs
  - *relifix* is less likely to introduce new regressions than *GenProg*

- Related questions:
  - How about regression in automatically generated patches?
  - How to avoid Regression Introducing Patches?
Search-Based Program Repair

Search-Based Repair Tools

Patch Generation

Candidate Patches

How do the patches look like?

How do the tests look like?

Tests Fail

• contains at least one failing test

Final Patch
Search-Based Program Repair

Test Script

```bash
$command $argument1 $argument2
RETVAL=$?
[ $RETVAL -eq 0 ] && echo Success
[ $RETVAL -ne 0 ] && echo Failure
```

Patch Evaluation

Tests

Check exit status of command
Non-zero exit status denotes test failure

- exit(-2);
Repair patterns from human patches

**Human patches**

**Automatic Program Repair**

### Anti-patterns

Set of generic forbidden transformations that can be enforced on top of any search-based repair tool.
Problem: Weak Oracle

Failing Test Script

```bash
$command $argument1 $argument2
RETVAL=$?
[ $RETVAL -eq 0 ] && echo Success
[ $RETVAL -ne 0 ] && echo Failure
```

- Statements like exit call/assertions serve as test proxies
- Test proxies should *not be* randomly manipulated

**A1: Anti-delete CFG exit node**

- Remove return statements, exit calls, functions with the word “error”, assertions.

```c
static void BadPPM(char* file) {
    fprintf(stderr, "%s: Not a PPM file.\n", file);
    exit(-2);
}
```
Problem: Inadequate Test Coverage

- Repair tools allow removal of code as long as all test passes
- Statements are mistakenly considered as redundant code
- Anti-patterns:
  - A2: Anti-delete Control Statement
  - A3: Anti-delete Single-statement CFG
  - A4: Anti-delete Set-Before-If

**A2: Anti-delete Control Statement**

× Remove control statements (e.g., if-statements, switch-statements, loops).

```cpp
  call_result = call_user_function_ex(...);
  - if (call_result == SUCCESS && ...) {
    - if (SUCCESS == statbuf_from_array(...))
    -   ret = 0;
    - } else if (call_result == FAILURE) {...
```
Problem: Non-termination

- Automatically generated patches may incorrectly remove loop update
  - Cause infinite loop

**A5: Anti-delete Loop-Counter Update**

⚠️ Remove assignment statement $A$ inside loop $L$ if:

\[
\{\text{Var in Termination Condition of } L\} \cap \{\text{Var in LHS of assignment } A\} = \emptyset
\]

```c
while( x> 5)
    - x++;`
Problem: Trivial Patch

- Trivial patch – patch that insert return-statements based on expected output
  Ex: `+if(test1)
      + return out1`

A6: Anti-append Early Exit

- Insert return/goto statement at any location except for after the last statement in a CFG node.

```c
+ if ((type != 0))
+ return;
zend_error((1<<3L),"Uninitialized string offset:","...);
```
Problem: Functionality Removal

- Removes functionality by inserting T/F

A7: Anti-append Trivial Conditions

- Insert trivial condition.
  - A condition is trivial if and only if it is:
    1) True/False Constant
    2) Tautology/Contradiction in expression (e.g., if(x || y || !y))
    3) Static analysis (e.g., if(x || y != 0), y is initialized)

```c
- if ((fmap[j].key != format->ptr[i + 1]))
+ if ((fmap[j].key != format->ptr[i + 1]) && !(1))
    continue;
```
Integrating Anti-patterns

Search-Based Repair Tools

Patch Generation

Candidate Patches

Is Anti-pattern?

YES

NO

Patch Evaluation

Tests Fail

Tests
• contains at least one failing test

All Tests Pass

Final Patch

YES

NO
How could anti-pattern helps?

- Evaluated on 12 open source projects
  - Enforcing anti-patterns leads to patches with better fix localization and delete less functionality.
  - Tools integrated with anti-patterns generate patches faster due to repair space reduction.

- Related questions:
  - Are existing program repair techniques effective in generating patches?
    - Anti-patterns reveal many problems in automatically generated patches
  - How about anti-patterns for repair operators? Could we get rid of repair operators that are ineffective?
Design of Repair Operators: Codeflaws

Programming Competition Benchmark for Objective Evaluation of Program Repair
**Codeflaws Benchmark**

- Obtained from Codeforces online database
- Diverse types of defects
  - 40 defects types
- Large number of defects
  - 4085 real defects
- Large number of programs
  - 7945 programs
- Large Held-out test suite for patch validation
  - 5-350 tests, Average: 40
- Non-trivial programs (algorithmically complex)
- Support large-scale controlled Experiments
- [https://codeflaws.github.io/](https://codeflaws.github.io/)
# Frequency and Effectiveness of Repair Operators

<table>
<thead>
<tr>
<th>Repair Operator</th>
<th>GenProg</th>
<th>SPR</th>
<th>Prophet</th>
<th>Angelix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Freq(%)</td>
<td>Eff(%)</td>
<td>Freq(%)</td>
<td>Eff(%)</td>
</tr>
<tr>
<td>Delete Statement</td>
<td>17.53</td>
<td>41.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert Assignment</td>
<td>17.39</td>
<td>38.46</td>
<td>5.77</td>
<td>43.10</td>
</tr>
<tr>
<td>Insert If</td>
<td>16.92</td>
<td>38.74</td>
<td>7.96</td>
<td>50.00</td>
</tr>
<tr>
<td>Loosen /Tighten Condition</td>
<td>54.53</td>
<td>22.35</td>
<td>46.06</td>
<td>19.95</td>
</tr>
<tr>
<td>Variable Replacement</td>
<td>8.51</td>
<td>56.73</td>
<td>6.46</td>
<td>29.36</td>
</tr>
<tr>
<td>Relational Operator Replacement</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*High frequency, Low Effectiveness*
Future Research

- Applications of Program Repair
  - Test-Driven Merging
    - Instead of using Longest Common Subsequence, use tests to drive merging of multiple programs
      - Provide additional guarantee that merged program pass all tests

- Anti-pat terns beyond Program Repair
  - Anti-patterns as specification for guiding repair
  - Anti-patterns as selected “code smells”
  - Adapt anti-patterns to other search-based software engineering activities (e.g., specific code anti-patterns identifying energy hot-spots)