Automatic, Efficient, and General Repair of Software Defects using Lightweight Program Analyses

Dissertation Proposal
Claire Le Goues
September 22, 2010
Software Errors Are Expensive

“Everyday, almost 300 bugs appear [...] far too many for only the Mozilla programmers to handle.”
– Mozilla Developer, 2005

• Even security-critical errors take 28 days to fix.
• Software errors in the US cost $59.5 billion annually (0.6% of GDP).

Proposed Solution

Automatic Error Repair
Previous Work

• Runtime monitors + repair strategies [Rinard, Demsky, Smirnov, Keromytis].
  – Increases code size, or run time, or both.
  – Predefined set of error and repair types.

• Genetic programming [Arcuri].
  – Proof-of-concept, limited to small, hand-coded examples.

• Lack of *scalability* and *generality*.
Insights

1. Existing program code and behavior contains the seeds of many repairs.
2. Test cases scalably provide access to information about existing program behavior.
Proposal

Use search strategies, test cases, and lightweight program analyses to quickly find a version of a program that doesn’t contain a particular error, but still implements required functionality.
Outline

• Repair technique metrics
• System overview
• Four research contributions, including preliminary results
• Schedule
• Conclusions
Overall Metrics

• **Scalability**
  – Lines of code. Success: hundreds of thousands of lines.
  – Time. Success: minutes.

• **Generality**
  – Varied benchmark set.
  – As much as possible, real programs (open source) with real vulnerabilities (public vulnerability reports).

• **Correctness**
  – Large, held-out test suites.
  – Performance on workloads.
FAR FROM GOAL:
DISCARD INPUT OUTPUT
EVALUATE DISTANCE BETWEEN EACH VARIANT AND GOAL
MUTATE TO CREATE NEARBY VARIANTS CLOSER TO GOAL: KEEP TRYING
FAR FROM GOAL: DISCARD
OUTPUT
Four Proposed Contributions

1. **Initial prototype**, with baseline representation, localization, and variant evaluation choices.

2. **Fault and fix localization**: Identify code implicated in the error (that might profitably be changed), and code to use to make changes.

3. **Repair templates**: Generalize previous work by mining and using repair templates, or pieces of code with “holes” for local variables.

4. **Precise objective function**: Develop a precise way to estimate the distance between a variant and a program that passes all test cases.
# Preliminary Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Size (loc)</th>
<th>Fault</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>example</td>
<td>22</td>
<td>Infinite loop</td>
<td>149 s</td>
</tr>
<tr>
<td>zune</td>
<td>example</td>
<td>28</td>
<td>Infinite loop</td>
<td>42 s</td>
</tr>
<tr>
<td>uniq</td>
<td>Text processing</td>
<td>1146</td>
<td>Segmentation fault</td>
<td>32 s</td>
</tr>
<tr>
<td>look-ultrix</td>
<td>Dictionary lookup</td>
<td>1169</td>
<td>Segmentation fault</td>
<td>42 s</td>
</tr>
<tr>
<td>look-svr4</td>
<td>Dictionary lookup</td>
<td>1363</td>
<td>Infinite loop</td>
<td>51 s</td>
</tr>
<tr>
<td>units</td>
<td>Metric conversion</td>
<td>1504</td>
<td>Segmentation fault</td>
<td>107 s</td>
</tr>
<tr>
<td>deroff</td>
<td>Document processing</td>
<td>2236</td>
<td>Segmentation fault</td>
<td>129 s</td>
</tr>
<tr>
<td>nullhttpd</td>
<td>webserver</td>
<td>5575</td>
<td>Remote heap overflow</td>
<td>502 s</td>
</tr>
<tr>
<td>indent</td>
<td>Code processing</td>
<td>9906</td>
<td>Infinite loop</td>
<td>533 s</td>
</tr>
<tr>
<td>flex</td>
<td>Lexer generator</td>
<td>18775</td>
<td>Segmentation fault</td>
<td>233 s</td>
</tr>
<tr>
<td>atris</td>
<td>Graphical tetris game</td>
<td>21553</td>
<td>Local stack overflow</td>
<td>69 s</td>
</tr>
<tr>
<td><strong>Total/Avg</strong></td>
<td></td>
<td><strong>63K</strong></td>
<td><strong>Infinite loop</strong></td>
<td><strong>171.7 s</strong></td>
</tr>
</tbody>
</table>
FAR FROM GOAL: DISCARD

EVALUATE DISTANCE BETWEEN EACH VARIANT AND GOAL

MUTATE TO CREATE NEARBY VARIANTS

CLOSER TO GOAL: KEEP TRYING

INPUT

OUTPUT
Four Proposed Contributions

1. **Initial prototype**, with baseline representation, localization, and variant evaluation choices.

2. **Fault and fix localization**: Identify code implicated in the error (that might profitably be changed), and code to use to make changes.

3. **Repair templates**: Generalize previous work by mining and using repair templates, or pieces of code with “holes” for local variables.

4. **Precise objective function**: Develop a precise way to estimate the distance between a variant and a program that passes all test cases.
Mutating a Program

• Given program A1:
  – With some probability, choose code at a location.
  – Insert code before it, or replace it entirely, by copying code from elsewhere in the same program, chosen with some probability.

• Result: program A2

Search space size is approximated by combining these probabilities over the entire program (how much we can change * how many ways we can change it).

Fault localization defines probability that code at a location is modified.
  • Goal: Code likely to affect bad behavior without affecting good behavior = high change probability

Fix localization defines probability that code is selected for insertion.
  • Goal: code likely to affect repair = high probability of selection.
Fault and Fix Localization: Idea

• Plan: use **machine learning** to relate lightweight features to fault/fix probability.
  – Statistics relating statements and dynamic data values to important events, like failure.
  – Static features shown by previous work to correlate with quality.

• Identify code that might affect variables implicated in failure, or code that is similar, but not identical, to likely-faulty code (the same, but includes a null-check, for example).
Fault and Fix Localization: Evaluation

• Effect on search space size (scalability):
  – *Score* metric: proportion of code eliminated from consideration (higher is better).
  – Measure space size by summing returned probability over the entire program (lower is better)

• Find/create benchmarks with difficult-to-localize errors, like SQL injection attacks (generality).
Four Proposed Contributions

1. **Initial prototype**: with baseline representation, localization, and variant evaluation choices.

2. **Fault and fix localization**: Identify code implicated in the error (that might profitably be changed), and code to use to make changes.

3. **Repair templates**: Generalize previous work by mining and using repair templates, or pieces of code with “holes” for local variables.

4. **Precise objective function**: Develop a precise way to estimate the distance between a variant and a program that passes all test cases.
```c
void gcd(int a, int b) {
    if (a == 0)
        printf("%d", b);
    else
        while (b > 0) {
            if (a > b)
                a = a - b;
            else
                b = b - a;
        }
    printf("%d", a);
    return;
}
```
Repair Templates: Idea

1. Mine promising template candidates from existing source code or the source control repository.

2. Synthesize templates from candidates, generating code with annotated “holes.”

3. Use a template to do mutation, as in previous work in error repair or dynamic compilation techniques.

```c
1. int gcd2(int a, int b) {
2.   if (a == 0) {
3.     printf("%d", b);
4.   while (b > 0) {
5.     if (a > b) {
6.       a = a - b;
7.     } else {
8.       b = b - a;
9.     }
10.    printf("%d", a);
11.   } return a;
12.}
```
Repair Templates: Evaluation

• Measure proportion of intermediate variants that compile (more is better).

• Formalize: small-step contextual semantics (optional).

• Find/create benchmarks with errors amenable to templated repairs (i.e.: errors handled in previous error repair work or repaired in the source code history).
EVALUATE DISTANCE BETWEEN EACH VARIANT AND GOAL

MUTATE TO CREATE NEARBY VARIANTS

FAR FROM GOAL: DISCARD

CLOSER TO GOAL: KEEP TRYING

INPUT

OUTPUT
Four Proposed Contributions

1. **Initial prototype**, with baseline representation, localization, and variant evaluation choices.

2. **Fault and fix localization**: Identify code implicated in the error (that might profitably be changed), and code to use to make changes.

3. **Repair templates**: Generalize previous work by mining and using repair templates, or pieces of code with “holes” for local variables.

4. **Precise objective function**: Develop a precise way to estimate the distance between a variant and a program that passes all test cases.
Evaluating Intermediate Variants

• The **objective function** estimates the distance between an intermediate variant and the goal (i.e., to pass all test cases); variants closer to goal are used in the next mutation round.

• Natural **baseline**: how many test cases does a variant pass?
A Buffer Underflow Vulnerability

1. void broken(int sock) {
2.   char* line, buff=NULL;
3.   int len;
4.   sgets(line,socket);
5.   len = atoi(line);
6.   // no bounds check
7.   buff=calloc(len * 2);
8.   // vulnerable recv
9.   recv(sock,buff,len);
10.  return buff;
11.}

1. void fixed(int sock) {
2.   char* line, ff=NULL;
3.   int len;
4.   sgets(line,socket);
5.   len = atoi(line);
6.   if(len>0 && len<MAX){
7.     buff=malloc(len * 2);
8.     recv(sock,buff,len);
9.   }
10.  return buff;
11.}
Objective Function: Idea

- Function should be **precise**, correlating well with actual distance; counting test cases is **imprecise** because it throws away intermediate information.
- Plan: use **machine learning** to relate differences in dynamic behavior between broken program and intermediate program to distance.

```c
1. void almost(int sock) {
2.     char* line, ff=NULL;
3.     int len;
4.     sgets(line,socket);
5.     len = atoi(line);
6.     if(len>0 && len<MAX){
7.         buff=calloc(len * 2);
8.         recv(sock,buff,len);
9.     }
10.    len = 5 / 0;
11.    return buff;
12.}
```
Objective Function: Evaluation

- Starting points for “actual” distance: tree-structured differencing, profiles of dynamic behavior.
- Estimate the function’s **fitness distance correlation**, or the correlation between it and the “ground truth”.
- Find/create benchmarks that require more than one change to repair.
• Graduate May 2013 (3 more years).
• Journal article on contribution 1 under revision.
• Slack in schedule: another internship, collaborative project on safety-critical medical equipment software, new ideas that arise from proposed research.
Conclusions

• Goal: scalable, general, correct automatic error repair.
• Approach: search closely-related programs for a version that passes all of the test cases.
• Questions to be answered:
  – What representation choices are necessary to make this possible? (Initial Prototype)
  – How should intermediate variants be created from nearby programs? (localization, templates)
  – How should intermediate variants be evaluated, to effectively guide the search? (Precise objective functions)


**Conference**


**Workshop**


Please ask difficult questions.