SemFix: Program Repair via Semantic Analysis

Ye Wang, PhD student
Department of Computer Science
Virginia Tech
Problem Statement

• Debugging takes much time and effort
• Even after root cause of a bug is identified, fixing bug is non-trivial
• Problem solved by this paper is how to automatically repair bugs
Example

To be fixed

Constraint

let bias = f(inhibit, up_sep, down_sep)
so that bias > down_sep can pass tests

Synthesize f
(1) try a constant:
cannot satisfy constraint

(2) try to use "+", \{v1+c, v1+v2\}:
f = up_sep + 100
Background

• Statistical fault localization
  • Localize root-cause of program failure by exploiting the correlation between execution of faulty statements and program failure

• Component-based program synthesis
  • Generate a program that satisfies all the given input-output pairs.
Approach

• Only generate a repair by altering one statement. The generated fix is always with respect to a given test suite.
  • Generate repair constraint
  • Generate a fix
Approach

• Generate repair constraint
  • The paper focuses on repairs changing the right side of assignments or branch predicates
    • \( x = f_{\text{buggy}}(\ldots) \rightarrow x = f(\ldots) \)
    • \( \text{if}(f_{\text{buggy}}(\ldots)) \rightarrow \text{if}(f(\ldots)) \)
  • No side effect: \( f(\ldots) \) do not modify any program variable

\text{Definition 1 (Repair Constraint):} Given a program \( P \), a test suite \( T \), a repair constraint \( C \) of a function \( f_{\text{buggy}} \) in program \( P \) is a constraint over function \( f \) such that if \( f \models C \), \( P[f/f_{\text{buggy}}] \) passes all tests in \( T \).

• Repair constraint \( C \) is a conjunction of constraints derived from \( T \). For each test \( t_i \), there is a constraint \( C_i \),
  \[ C = \bigwedge_{i=1}^{n} C_i \]
Approach

• Generate repair constraint
  • Each $C_i$ is a predicate over the function $f$
  • To generate $C_i$, the paper uses symbolic execution in a novel fashion.
  • Traditional symbolic execution takes all input variables as symbolic, while the paper's symbolic execution starts with a concrete input.
  • Execute the program concretely with input $t_i$ to statement $s$. Denote the program state before executing statement $s$ as $\xi_i$. Then set the result of function $f(...)$ as symbolic and continue symbolic execution from statement $s$. 
Approach

• Generate repair constraint
  • $\tau_i$: symbolic value assigned to result of function $f(\ldots)$
  • Symbolic execution explores $m$ paths.
  • For each path $\pi_j$, $pc_j$ is the associated path condition, and $O_j$ is the symbolic expression of output
  • $O(t_i)$ is the expected output of program $P$ with input $t_i$
  • Constraint: $C_i := (\bigvee_{j=1}^{m} (pc_j \land O_j == O(t_i))) \land (f(\xi_i) == \tau_i)$

• First part means at least one feasible path along which output of program $P$ is the same as the expected output.
• Second part builds up input-output relationship of function $f$
Approach

• Generate repair constraint

<table>
<thead>
<tr>
<th>Test</th>
<th>inhibit</th>
<th>up_sep</th>
<th>down_sep</th>
<th>Expected output</th>
<th>Observed output</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>pass</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>110</td>
<td>1</td>
<td>0</td>
<td>fail</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>100</td>
<td>50</td>
<td>1</td>
<td>1</td>
<td>pass</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>-20</td>
<td>60</td>
<td>1</td>
<td>0</td>
<td>fail</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>pass</td>
</tr>
</tbody>
</table>

\( C_i := (\bigvee_{j=1}^m (p_{c_j} \land O_j == O(t_i))) \land (f(\xi_i) == \tau_i) \)

\((X > 110 \land 1 = 1) \lor (X \leq 110 \land 1 = 0)\)
can be simplified to \(X > 110\)

\(f(1,11,110) = X\)

\(f(1,11,110) > 110\)
Approach

• Generate a fix
  • Component based program synthesis
  • Input-output pairs of to-be-synthesis program are encoded into constraints on a set of location variables $L$, a valuation of which leads to a program that satisfies the given input-output pairs.
  • Constraint $\psi_{func}(L, \alpha, \beta)$ dictates that the synthesized program must produce output $\beta$ when given input $\alpha$
  • Input-output pair $\langle \xi_i^k, \tau_i^k \rangle$ is generated when $f$ is hit at the $k$th time in the execution of program $P$ with input $t_i$, but $\langle \xi_i^k, \tau_i^k \rangle$ is symbolic in terms of $\{\tau_i^k | 1 \leq k \leq \omega\}$, where $\omega$ is number of times $f$ is executed with input $t_i$
Approach

• Generate a fix
  \( \{ \tau_i^k | 1 \leq k \leq \omega \} \) satisfy \( (\bigvee_{j=1}^m (pc_j \land O_j == O(t_i))) \)
  \( f \) should satisfy the constraint

\[
\theta_i \overset{\text{def}}{=} \exists \tau_i, \bigwedge_{k=1}^w \phi_{func}(L, \xi_i^k, \tau_i^k) \land (\bigvee_{j=1}^m (pc_j \land O_j == O(t_i)))
\]

where \( \tau_i := \{ \tau_i^k | 1 \leq k \leq w \} \).

• Conjoin constraints from all tests together with the well-formedness constraint \( \psi_{\omega fp} \)

\[
\theta \overset{\text{def}}{=} (\bigwedge_{i=1}^n \theta_i) \land \psi_{\omega fp}(L)
\]
Approach

• Putting it all together
  • The algorithm takes as inputs a buggy program P, a test suite T and a ranked list of suspicious program statements RC
  • When successful, the algorithm produces a repair, applying which on P makes P pass all tests in the test suite T.

<table>
<thead>
<tr>
<th>Level</th>
<th>Conditional Statement</th>
<th>Assign Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constants</td>
<td>Constants</td>
</tr>
<tr>
<td>2</td>
<td>Comparison (&gt;, ≥, =, ≠)</td>
<td>Arithmetic (+, −)</td>
</tr>
<tr>
<td>3</td>
<td>Logic (∧, ∨)</td>
<td>Comparison, Ite</td>
</tr>
<tr>
<td>4</td>
<td>Arithmetic (+, −)</td>
<td>Logic</td>
</tr>
<tr>
<td>5</td>
<td>Ite, Array Access</td>
<td>Array Access</td>
</tr>
<tr>
<td>6</td>
<td>Arithmetic (⋆)</td>
<td>Arithmetic (⋆)</td>
</tr>
</tbody>
</table>
Evaluation

• Use SemFix to repair seeded defects and real defects in an open source software. The proposed method is also compared with genetic programming based repair techniques.

<table>
<thead>
<tr>
<th>Subject Prog.</th>
<th>Size (LOC)</th>
<th>#Versions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tcas</td>
<td>135</td>
<td>41</td>
<td>air-traffic control program</td>
</tr>
<tr>
<td>Schedule</td>
<td>304</td>
<td>9</td>
<td>process scheduler</td>
</tr>
<tr>
<td>Schedule2</td>
<td>262</td>
<td>9</td>
<td>process scheduler</td>
</tr>
<tr>
<td>Replace</td>
<td>518</td>
<td>29</td>
<td>text processor</td>
</tr>
<tr>
<td>Grep</td>
<td>9366</td>
<td>2</td>
<td>text search engine</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>90</strong></td>
<td></td>
</tr>
</tbody>
</table>
Evaluation

• Intuitively, it is harder to generate a repair to pass more tests

• Repairs generated with small number of tests may not be valid for other tests that are not in test suite.

Comparing the success rate between SemFix (SF) and GenProg (GP). X in [X] on the top of each column denotes the number of tests.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tcas</td>
<td>38 / 24</td>
<td>38 / 19</td>
<td>35 / 16</td>
<td>34 / 12</td>
<td>34 / 11</td>
</tr>
<tr>
<td>Schedule</td>
<td>5 / 1</td>
<td>3 / 1</td>
<td>4 / 1</td>
<td>4 / 0</td>
<td>4 / 0</td>
</tr>
<tr>
<td>Schedule2</td>
<td>4 / 4</td>
<td>3 / 2</td>
<td>4 / 2</td>
<td>3 / 3</td>
<td>2 / 1</td>
</tr>
<tr>
<td>Replace</td>
<td>7 / 6</td>
<td>7 / 5</td>
<td>8 / 5</td>
<td>7 / 6</td>
<td>6 / 4</td>
</tr>
<tr>
<td>Grep</td>
<td>2 / 0</td>
<td>1 / 0</td>
<td>1 / 0</td>
<td>2 / 0</td>
<td>2 / 0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>56 / 35</strong></td>
<td><strong>52 / 27</strong></td>
<td><strong>52 / 24</strong></td>
<td><strong>50 / 21</strong></td>
<td><strong>48 / 16</strong></td>
</tr>
</tbody>
</table>
Evaluation

Fig. 4. Comparing the running time between SEMFix and GenProg.
Evaluation

- Different types of bugs

**SemFix (SF) vs. GenProg (GP) in repairing different class of bugs with 50 tests.**

<table>
<thead>
<tr>
<th>Bug type</th>
<th>Const</th>
<th>Arith</th>
<th>Comp</th>
<th>Logic</th>
<th>Code Missing</th>
<th>Redundant Code</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>14</td>
<td>14</td>
<td>16</td>
<td>10</td>
<td>27</td>
<td>9</td>
<td>90</td>
</tr>
<tr>
<td>SemFix</td>
<td>10</td>
<td>6</td>
<td>12</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>48</td>
</tr>
<tr>
<td>GenProg</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>16</td>
</tr>
</tbody>
</table>
Related Work

• Genetic programming:

• AutoFix-E and AutoFix-E2 are based on the program contracts in Eiffel programs

• Jobstmann, et. al. uses LTL specifications for finite state programs
Related Work

- Gopinath, et. al. use behavioral specifications and encode the specification constraint on the buggy program into SAT constraint.
- Robert and Roderick employ template based repair for linear expressions.
- Dallmeier, et. al. try to generate fixes from object behavior anomalies.
- ClearView follows a similar scheme but works on deployed binary program when high availability is required.
Related Work

• BugFix suggests bug-fix that has been used in a similar debugging situation.
• Debroy and Wong propose to use mutation for program repair.
• PHPRepair focuses on HTML generation errors in PHP programs.
• Instead of fixing a buggy program, program sketching allows a programmer to write a sketch of the implementation idea while leaving the low level details omitted as holes to be automatically filled up by the sketch complier.
Conclusion

• The proposed SemFix is a semantics based program repair tool.
• The *repair constraint*, which is derived from a set of tests, is solved by generating a valid repair.
• SemFix can synthesize a repair even if the repair code does not exist anywhere in the program.
Discussion

• Which is easier, fixing a bug manually or verify the auto-generated bug fix?
Discussion

• Can you apply artificial intelligence (AI) to automatic bug repairing to improve it?
• If yes, how?
Discussion

• If you are a software engineer in an IT company, will you use an automatic bug repairing tool?
• If yes, which cases will you use it in? which cases will you not use it in?
Discussion

• The paper says "the test suite could be large and thus affect the scalability of our technique".
• Do you think selecting a subset of the entire test suite for repair generation is a good idea?
Discussion

• These basic components are used to generate a repairs.

• Do you think they are enough? Should we add something more, like division (/)?

<table>
<thead>
<tr>
<th>Level</th>
<th>Conditional Statement</th>
<th>Assign Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constants</td>
<td>Constants</td>
</tr>
<tr>
<td>2</td>
<td>Comparison (&gt; ≥ = ≠)</td>
<td>Arithmetic (+, −)</td>
</tr>
<tr>
<td>3</td>
<td>Logic (∧, ∨)</td>
<td>Comparison, Ite</td>
</tr>
<tr>
<td>4</td>
<td>Arithmetic (+, −)</td>
<td>Logic</td>
</tr>
<tr>
<td>5</td>
<td>Ite, Array Access</td>
<td>Array Access</td>
</tr>
<tr>
<td>6</td>
<td>Arithmetic (*)</td>
<td>Arithmetic (*)</td>
</tr>
</tbody>
</table>
Discussion

• The proposed method only synthesis an expression.
• Should we use some more complicated logics, like if-condition, for-loop, and while-loop?
• If yes, how will they affect the precision and speed of the bug repairing method?
Discussion

• The proposed method only change one statement.
• Do you think changing more statements is a good idea? Why?
Discussion

• To be honest, no matter how many test cases are used, we can not guarantee the bug fix is right.
• Can the bug repairing method use another constraint, instead of tests?
Discussion

• Can any other research be done based on SemFix?
• If yes, talk about the details.