Paper Review On

Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis

By Sandeep Polisetty, Xiaoyi Duan, Yuzhi Xiao
Introduction

Automated repair tools today: GenProg, PAR, relifix, Semfix and so on.
Search-based vs semantics-based

Search base: GenProg, PAR and SPR
Semantic-base: SemFix, Nopol and Direct Fix
Important attributes for automated program repair

- Scalability
- Repairability
- Quality of repair
Balance and trade-offs between two types of repairs

SPR (search-based repair) generates more repairs and has good scalability.

Semantic-based repair has high repair quality but low scalability.

But for Angelix, it can also scale up to the same level as the most advanced search-based repair tool, using lightweight repair constraint, angelic forest and fault localization.
How does this semantic-based repair scale?

The repair constraint, angelic forest, has a size that is independent of the size of the program.
Phases of Program Repair

1. Semantic Transformation to expand class of defects repaired
2. Fault localization through heuristic based identification of possible suspicious expressions and added symbols in place
3. Constraint Generation
4. Repair generation which satisfies constraint
Example

Buggy Problem:
Int modulus_subtract(a,b):
    If (a>b) {
        Return a-b
    }
Else{
    Return a-b;
}

Correct Problem
Int modulus_subtract(a,b):
    If (a>b) {
        Return a-b
    }
Else{
    Return b-a;
}
Going to Step 2 Directly (Fault Localization)

Hierustic: All Statement assignments

Buggy Problem:
Int modulus_subtract(a,b):
   If (a>b) {
      Return alpha;
   }
   Else{
      Return Beta;
   }

Test Cases
T1: <a=1,b=2,out=1>
T2:<a=2,b=1,out=1>
Evaluate modified algorithm for each testcase

Path at the point of each symbol evaluation
Is the snapshot of all visible variables

\{a: 1, b: 2\} for test case 1
Constraint Equation generation

Algorithm 1 Angelic forest generation

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{Input:} program $P$, test case $(I, O_e)$
\State \textbf{Input:} a set of suspicious expressions $E$
\State \textbf{Output:} angelic forest $A$
\While{there is an unexplored path $\land \neg$timeout}
\State // perform controlled symbolic execution
\State $(pc, O_a) \leftarrow \text{CONTROLLEDSYMEXE}(I, E)$
\State $R \leftarrow pc \land O_a = O_e$
\If{$R$ is satisfiable}
\State $M \leftarrow \text{GETMODEL}(R)$ // via a constraint solver
\State $A \leftarrow A \cup \text{EXTRACTANGELICPATH}(M)$
\EndIf
\EndWhile
\State \Return $A$
\end{algorithmic}
\end{algorithm}

\begin{itemize}
\item $\Rightarrow (\alpha = 1) \land (x=1) \land (y=2)$
\end{itemize}

This can be satisfied and is appended to the angelic path of the test t1.

If this constraint can’t be satisfied then it is implied that modifying these symbols would not pass the testcase.
**Angelic Forest**

Angelic Value of suspicious expression; The value of the expression that passes a particular test case

For testcase \{a:1,b:2\} -> alpha:1

Angelic Path: The list of all tuples \langle Symbol, Angelic Value. Snapshot of Visible variables \rangle. The list of the symbol and the snapshot of all variables when a symbol is evaluated (to aid in the selection of components)

Test Case 1: \{alpha,1,\{a:1,b:2\}\},

Test Case 2: \{Beta,1,\{a:2,b:1\}\}

Angelic Forest -> \{Test Case 1: Path 1, Path 2 \}, \{Test Case 2: Path 3\}
Repair Generation

\[
\bigvee_{\pi_2(t,E)} \bigwedge_{(e^k,v,\sigma)} \left( \bigwedge_{x \in \text{dom}(\sigma)} x[e^k] = \sigma(x) \land e^k = v \right),
\]

\((x = 1) \land (y = 2) \land (\alpha = 1)) \land \\
\((x = 2) \land (y = 1) \land (\beta = 1))

The generated repair must satisfy at least one path for each test case and component must be chosen to satisfy the output with the visible variable chosen as input states.
Scalability??

Size of the forest is a function of number of suspicious expressions $N$ and not of program size.

Multi line fixes can be repaired as the impact of one repair on another is captured through the constraint equation.

Can be started with a small subset of the testsuite which is later expanded.

It does not explore cases where there is no Angelic Path.
Experimental Results

RQ1. Can our repair method generate repairs from large-scale real world software

RQ2. Can our repair method fix multi-location bugs?
<table>
<thead>
<tr>
<th>Subject</th>
<th>LoC</th>
<th>Tests</th>
<th>Versions</th>
<th>W/I Our Defect Class</th>
<th>Fixed Defects</th>
<th>Equiv. to Developer Fixes</th>
<th>Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Angelix</td>
<td>SPR</td>
<td>GenProg</td>
</tr>
<tr>
<td>wireshark</td>
<td>2814K</td>
<td>63</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>php</td>
<td>1046K</td>
<td>8471</td>
<td>44</td>
<td>12</td>
<td>10</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>gzip</td>
<td>491K</td>
<td>12</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>gmp</td>
<td>145K</td>
<td>146</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>libtiff</td>
<td>77K</td>
<td>78</td>
<td>24</td>
<td>12</td>
<td>10</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Overall</td>
<td>82</td>
<td>32</td>
<td>28</td>
<td>31</td>
<td>11</td>
<td>19</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2: Experimental results
Comparison with other tools

Repairability

Repair quality

Multi-location bugs

Angelix is not only scalable but also less frequently generates functionality-deleting repairs than the existing tools such as SPR and GenProg.

Only repair tool for generating (non-functionality-deleting) fixes for multi-location bugs in large-scale real-word software
Table 3: The number of defects exclusively repaired by each repair tool across the subjects

<table>
<thead>
<tr>
<th>Subject</th>
<th>Angelix</th>
<th>SPR</th>
<th>GenProg</th>
<th>AE</th>
</tr>
</thead>
<tbody>
<tr>
<td>wireshark</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>php</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>gzip</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>gmp</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libtiff</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Overall</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

1. if (td->td_nstrips > 1 && td->td_compression == COMPRESSION.NONE && td->td_stripbytecount[0] != td->td_stripbytecount[1])

(a) The buggy location of libtiff-d13be72c-ccadf48a

1. if (td->td_nstrips > 2 && td->td_compression == COMPRESSION.NONE && td->td_stripbytecount[0] != td->td_stripbytecount[1])

(b) The repair generated by our tool, Angelix

1. if (td->td_nstrips > 1 && td->td_compression == COMPRESSION.NONE && td->td_stripbytecount[0] != td->td_stripbytecount[1] && ll1)

(c) The repair generated by SPR
### Table 4: The number of functionality-deleting repairs

<table>
<thead>
<tr>
<th>Subject</th>
<th>Angelix</th>
<th></th>
<th>SPR</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Fixes</strong></td>
<td><strong>Del</strong></td>
<td><strong>Per</strong></td>
<td><strong>Fixes</strong></td>
</tr>
<tr>
<td>wireshark</td>
<td>4</td>
<td>1</td>
<td>25%</td>
<td>4</td>
</tr>
<tr>
<td>php</td>
<td>10</td>
<td>3</td>
<td>30%</td>
<td>18</td>
</tr>
<tr>
<td>gzip</td>
<td>2</td>
<td>0</td>
<td>0%</td>
<td>2</td>
</tr>
<tr>
<td>gmp</td>
<td>2</td>
<td>0</td>
<td>0%</td>
<td>2</td>
</tr>
<tr>
<td>libtiff</td>
<td>10</td>
<td>2</td>
<td>20%</td>
<td>5</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td><strong>28</strong></td>
<td><strong>6</strong></td>
<td><strong>21%</strong></td>
<td><strong>31</strong></td>
</tr>
</tbody>
</table>

### Table 5: Experimental results for multi-location defects.

<table>
<thead>
<tr>
<th>Defect</th>
<th>Fixed Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>libtiff-4a24508-cc79c2b</td>
<td>2</td>
</tr>
<tr>
<td>libtiff-829d8c4-036d7bb</td>
<td>2</td>
</tr>
<tr>
<td>coreutils-00743a1f-ec48bead</td>
<td>3</td>
</tr>
<tr>
<td>coreutils-1dd8a331-d461bfd2</td>
<td>2</td>
</tr>
<tr>
<td>coreutils-c5ccf29b-a04ddb8d</td>
<td>3</td>
</tr>
</tbody>
</table>
Heartbleed Bug

(a) The buggy part of the Heartbleed-vulnerable OpenSSL

```c
if (hbttype == TLS1_HB_REQUEST) {
  ...  
  memcpy(bp, pl, payload);
  ...  
}
```

(b) A fix generated by our tool, Angelix

```c
if (hbttype == TLS1_HB_REQUEST
    && payload + 18 < s->s3->rrec.length) {
  /* receiver side: replies with TLS1_HB_RESPONSE */
}
```

(c) The developer-provided repair

```c
if (1 + 2 + payload + 16 > s->s3->rrec.length)
  return 0;
...

else if (hbttype == TLS1_HB_RESPONSE) {
  /* sender side */
}
```

Figure 4: The Heartbleed bug and their fixes
Threats to Validity

1) Subject programs in the existing benchmark previously used to evaluate GenProg, AE and SPR. The validity of the experimental results are limited.

2) Configurations (the maximum number of suspicious locations)

3) Components particular to the chosen defect class
Conclusion

A semantic-based repair method

Novel lightweight repair repair constraint called ‘angelic forest’

Better repair quality

Successfully fixed multi-location bugs
Questions

1. By using the MaxSMT how can it guarantee minimal change, as the solver by definition would try to satisfy as many components as possible? This should increase the size of the change.

2. Would this work on composite components?

3. How is loop unrolling done for For loops?

4. Would the search space increase exponentially as the number of inputs to components increase?

5. When solving the constraint, can expressions that are satisfied by existing code removed from the constraint equation?
Thank you!