Enhancing Automated Program Repair With Deductive Verification

Xuan Bach D. Le\textsuperscript{1}, Quang-Loc Le\textsuperscript{2}, David Lo\textsuperscript{1}, Claire Le Goues\textsuperscript{3}

\textsuperscript{1}Singapore Management University
\textsuperscript{2}Singapore University of Technology and Design
\textsuperscript{3}Carnegie Mellon University
Automatic patch generation seeks to improve software quality.

- Bugs in software incur tremendous maintenance cost.
  
  In 2006, everyday, almost 300 bugs appear in Mozilla […] far too much for programmers to handle

- Developers presently debug and fix bugs manually.

- Automated program repair:

  APR = Fault Localization + Repair Strategies
Automatic patch generation seeks to improve software quality.

• Bugs in software incur tremendous maintenance costs.
• Developers presently debug and fix bugs manually.
• Automated program repair:

  1. Search: syntactic, or heuristic, “guess and check.”
  2. Semantic: symbolic execution + SMT solvers, synthesis.

APR = Fault Localization + Repair Strategies
KEY IDEA: COMBINE BOTH SEARCH- AND SEMANTICS-BASED REPAIR, WITH DEDUCTIVE VERIFICATION.

Benefits: more expressive than just one or the other, with correctness guarantees!
Specs

Verifier

Semantic candidates

Syntactic candidates

Faulty locations

Violated Specs

Yes!

No!

Stop

Genetic Programming

?
HIP/SLEEK: takes as input a buggy program and separation logic specification.

- Identifies components of spec that are violated.
- Localize to potentially implicated source locations/constructs:
  - Semantic: if- and loop-conditions (backwards dependency from later statements), right-hand-side of assignments.
  - Syntactic: statement level
- Verify correctness of candidate patched programs.
Example

```c
bool addint (int c, int[] out, int *j, int max)
{
    bool result = false;
    if ( *j >= max ) result = false;
    else{
        *j = *j + 1;
        out[*j] = c; //Bug: out array may overflow
        result = true;
    }
    return result;
}
```
Example

```c
bool addint (int c, int[] out, int *j, int max) {
    /* @Spec req j\rightarrow int_ref<j_val> & max \geq 0 & j_val \leq max 
    case { 
    j_val=max -> ens j\rightarrow int_ref<j_val> & j_val'=j_val & res=false
    j_val<max -> req j_val\geq 0 ens j\rightarrow int_ref<j_val> & j_val'=j_val+1 & out'[j_val'-1]=c & j_val'\leq max & res=true
    }*/
    {
        bool result = false;
        if( *j >= max ) result = false;
        else{
            *j = *j + 1;
            out[*j] = c; //Bug: out array may overflow
            result = true;
        }
    }
    return result;
}
```
Specification language: separation
Logic as supported by HIP/SLEEK

\[ \mathbf{Y} ::= \text{requires } \Phi \mathbf{Y} \mid \text{case}\{ \pi_1 \Rightarrow \mathbf{Y}_1; \ldots; \pi_n \Rightarrow \mathbf{Y}_n \} \mid \text{ensures } \Phi \]

- Example:

\[
\text{req } j \rightarrow \text{int_ref}<j_{\text{val}}> \& \text{max } \geq 0 \& j_{\text{val}} \leq \text{max} \\
\text{case } \{
\text{ens } j \rightarrow \text{int_ref}<j_{\text{val}}> \& j_{\text{val}}'=j_{\text{val}} \& \text{res}=\text{false} \\
j_{\text{val}}<\text{max } \rightarrow \\
\text{req } j_{\text{val}} \geq 0 \\
\text{ens } j \rightarrow \text{int_ref}<j_{\text{val}}> \& j_{\text{val}}'=j_{\text{val}}+1 \& \\
\text{ens } j_{\text{val}}' \leq \text{max} \& \text{out}'[j_{\text{val}}'-1]=c \& \text{res}=\text{true} 
\}
Specification language: separation
Logic as supported by HIP/SLEEK

\[ Y ::= \text{requires } \Phi \ Y \mid \text{case}\{\pi_1 \Rightarrow Y_1; \ldots; \pi_n \Rightarrow Y_n\} \mid \text{ensures } \Phi \]

- **Example:**

  req \( j \rightarrow \text{int_ref}<j\_val> \) & \( \text{max} \geq 0 \) & \( j\_val \leq \text{max} \)

  case {
  j\_val=\text{max} - \rightarrow
  ens \( j \rightarrow \text{int_ref}<j\_val> \) & \( j\_val'=j\_val \) & \( \text{res}=false \)
  j\_val<\text{max} - \rightarrow
  req \( j\_val \geq 0 \)
  ens \( j \rightarrow \text{int_ref}<j\_val> \) & \( j\_val'=j\_val+1 \) & \( j\_val' \leq \text{max} \) & \( \text{out'}[j\_val'-1]=c \) & \( \text{res}=true \)
  }

10
Example

```c
bool addint (int c, int[] out, int *j, int max)
/* @Spec req j\rightarrow int\_ref\langle j\_val\rangle \& max \geq 0 \& j\_val \leq max
 case {j\_val=max -> ens j\rightarrow int\_ref\langle j\_val\rangle \& j\_val’=j\_val \& res=false
 j\_val<max -> req j\_val\geq 0 ens j\rightarrow int\_ref\langle j\_val\rangle \& j\_val’=j\_val+1 \& out’[j\_val’-1]=c \& j\_val’\leq max \& res=true
 */
{
    bool result = false;
    if( *j \geq max ) result = false;
    else{
        *j = *j + 1;
        out[*j] = c; //Bug: out array may overflow
        result = true;
    }
    return result;
}
```
Example

```cpp
bool addint (int c, int[] out, int *j, int max)
/* @Spec req j⇒ int_ref<j_val> & max >=0 & j_val <= max
   case { j_val=max -> ens j⇒int_ref<j_val> & j_val'=j_val & res=false
       j_val<max -> req j_val>=0 ens j⇒int_ref<j_val> & j_val'=j_val+1 &
       out'[j_val'-1]=c & j_val'<=max & res=true }
*/
{
    bool result = false;
    if( *j >= max ) result = false;
    else{
        *j = *j + 1;
        out[*j] = c; //Bug: out array may overflow
    }
    return result;
}
```
Semantic Candidates via Violated Specs

• Identify relevant violated sub-formula
  – Preconditions, case blocks => expressions of if-condition
    
    ```
    case {
    j_val=max -> ... 
    j_val<max -> ... 
    }
    ```

  – Otherwise => assignment

    out'[j_val'-1]=c    out[*j -1]=c
Syntactic Candidates via statement-level operators.

- We use genetic programming to additionally generate syntactic candidates

- Mutation operators:
  - Delete: delete a statement
  - Replace: replace a statement by another
  - Swap: swap two statements
  - Append: append a statement after another

- This helps deal with general bugs
Example

```c
bool addstr (int c, int[] out, int *j, int max)
/* @Spec req j ⇒ int_ref<j_val> & max >=0 & j_val <= max 
case { 
j_val=max -> ens j ⇒ int_ref<j_val> & j_val'=j_val & res=false 
j_val<max -> req j_val>=0 ens j ⇒ int_ref<j_val> & j_val'=j_val+1 & 
out'[j_val'-1]=c & j_val'<=max & res=true 
} */
{ 
    bool result = false;
    if( *j >= max ) result = false;
    else{
        *j = *j + 1;
        out[*j] = c; //Bug: out array may overflow
    }
    return result;
}
```

Via semantic analysis

```
out[*j -1]=c
```
Candidates Selection via Verification

• **Recap**: condense search space with more valuable candidates, including semantics and syntactic candidates

• **Next**: verify, evolve candidates, and choose best ones
  – Use static verifier for modular verification
  – Fitness function: Select candidates with fewer warnings
  – Evolve until find one passing verification
## Experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>Mutated Loc</th>
<th>Loc</th>
<th>Time (minutes)</th>
<th>Bug Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniq</td>
<td>gline_loop</td>
<td>74</td>
<td>0.5</td>
<td>Incorrect</td>
</tr>
<tr>
<td>replace</td>
<td>addstr</td>
<td>855</td>
<td>2.8</td>
<td>Missing</td>
</tr>
<tr>
<td>replace</td>
<td>stclose</td>
<td>855</td>
<td>2.15</td>
<td>Missing</td>
</tr>
<tr>
<td>replace</td>
<td>stclose</td>
<td>855</td>
<td>2.2</td>
<td>Incorrect</td>
</tr>
<tr>
<td>replace</td>
<td>locate</td>
<td>855</td>
<td>2.5</td>
<td>Incorrect</td>
</tr>
<tr>
<td>replace</td>
<td>patsize</td>
<td>855</td>
<td>0.5</td>
<td>Incorrect</td>
</tr>
<tr>
<td>replace</td>
<td>esc</td>
<td>855</td>
<td>2.14</td>
<td>Incorrect</td>
</tr>
<tr>
<td>schedule3</td>
<td>dupp</td>
<td>693</td>
<td>0.43</td>
<td>Incorrect</td>
</tr>
<tr>
<td>print_tokens</td>
<td>ncl</td>
<td>1002</td>
<td>6.25</td>
<td>Missing</td>
</tr>
<tr>
<td>tcas2</td>
<td>IBC</td>
<td>302</td>
<td>0.15</td>
<td>Incorrect</td>
</tr>
</tbody>
</table>

Data: 10 seeded bugs from SIR benchmark  
Specifications written by second author of the paper
### Experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>Mutated Loc</th>
<th>Loc</th>
<th>Time (minutes)</th>
<th>Bug Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniq</td>
<td>gline_loop</td>
<td>74</td>
<td>0.5</td>
<td>Incorrect</td>
</tr>
<tr>
<td>replace</td>
<td>addstr</td>
<td>855</td>
<td>2.8</td>
<td>Missing</td>
</tr>
<tr>
<td>replace</td>
<td>stclose</td>
<td>855</td>
<td>2.15</td>
<td>Missing</td>
</tr>
<tr>
<td>replace</td>
<td>stclose</td>
<td>855</td>
<td>2.2</td>
<td>Incorrect</td>
</tr>
<tr>
<td>replace</td>
<td>locate</td>
<td>855</td>
<td>2.5</td>
<td>Incorrect</td>
</tr>
<tr>
<td>replace</td>
<td>patsize</td>
<td>855</td>
<td>0.5</td>
<td>Incorrect</td>
</tr>
<tr>
<td>replace</td>
<td>esc</td>
<td></td>
<td>2.14</td>
<td>Incorrect</td>
</tr>
<tr>
<td>schedule3</td>
<td>dupp</td>
<td></td>
<td>0.43</td>
<td>Incorrect</td>
</tr>
<tr>
<td>print_tokens</td>
<td>ncl</td>
<td>1002</td>
<td>6.25</td>
<td>Missing</td>
</tr>
<tr>
<td>tcas2</td>
<td>IBC</td>
<td>302</td>
<td>0.15</td>
<td>Incorrect</td>
</tr>
</tbody>
</table>

Data: 10 seeded bugs from SIR benchmark  
Specifications written by second author of the paper  

Angelix can only fix tcas2
Our Observations

• Angelix cannot deal with “missing implementation” bugs and is otherwise limited in the composition of its search space.

• Difference compared to our technique:
  – Angelix relies on test cases, which are an under-approximation of correctness requirements.
  – Our technique uses specs, which can express fully the desired behavior, but are less common in practice.
Conclusion

• We combine semantics-based and search-based APR via deductive verification

• We showed that:
  – Our technique fixes more bugs than state-of-the-art semantics-based APR, i.e. Angelix
  – Ensure repair soundness, mitigating overfitting.

• Future plans: automatically infer specs, experiment with different fitness functions…