A static analyzer for finding dynamic programming errors

2002. 11. 19

Traditional Check’s Problems

• for code generation, not for error-check
• Inter-procedural (interactions between function are not detected)
• reporting false errors. (non-achievable path)
• Require substantial additional work by the user
• requiring test cases

Goal

• Develop a source code analyzer that could find Purify-like errors with Purify’s ease of use, but without needing test cases.

Requirement

• C and C++ program should be checked effectively.
• Information should be derived from the program text rather than acquired through user annotations.
• Analysis should be limited to achievable paths
• The information produced from the analysis should be enough to allow a user to characterize the underlying defects easily.
**PREfix**

- Simulation: Use Virtual Machine => Memory Test & restrict achievable path.
- The behavior of a function: Model
  - Model is automatically generated by information extracted from programs.
- Bottom-Up: can apply an entire program, or subset of a program.

### The Advantage of Simulation

- Easily get achievable path
- Memory: exact values and predicates
  - Memory Layout
  - Bit Operation
  - Dereference
- End-of-path analysis
  Any unreachable memory or resource that has been allocated, but not freed, is reported.

(*disadvantage*)

achievable paths is often quite large -> samples of achievable paths to simulate. required a user config

### Pseudo-code for function simulation

```plaintext
while (there are more paths to simulate) {
  initialize memory state
  simulate the path, identifying inconsistencies and updating the memory state
  perform end-of-path analysis using the final memory state,
    identifying inconsistencies and creating per-path summary
}
combine per-path summaries into a model for the function
```


**Conditions, assumptions and choice points**

- 4 char *f(int size)
- 5 {
- 6     char *result;
- 7     
- 8     if (size <= 0) {
- 9     return "a";
- 10    }
- 11    result = malloc(size);
- 12    if (size == 1) { return NULL; }
- 13    return result;
- 14 }  
- 
- We don't know the value of 'size'.

- Assumption: size > 0
- Condition: 8: if (size > 0) => false
- Choice points: 10: if (size == 1) (need to make assumption: size == 1, size != 1)

- Assumption: size <= 0
- Condition: 8: if (size > 0) => false
- 10: if (size == 1) => false
- 12: if (size == 1) => false
- Choice points:

**Models**

- MODEL: The behavior of a function: model consists of exclusive outcomes.

**OUTCOME**

- **Guards** -> an element of test set (input dependent)
- **Constraint** -> precondition
- **Results** -> postcondition

**Ex. Model**

```
1  int deref (int *p) { 
2      (deref)
3      { (param p)
4        if (p == NULL) { (alternate return p)
5          return NULL; 
6          (constraint memory_initialized p)
7        }
8        (alternate return 3
9          (guard p == NULL)
10         (constraint memory_initialized p)
11         (constraint memory_valid_pointer p)
12         (constraint memory_initialized p)
13         (results p return p)
14       )
15     }
```

Figure 1. The model of the dereferencing function.
**Model Generation**

- System lib's models are provided by PREfix.
- **automatic** generation from user program
- **Merging states(outcomes)** at the end of function simulation is not necessary semantically but performance.

**If source is not available?**

- Third party components: source code or models are not available => cannot make models.
- The general philosophy: **avoid generating incorrect messages** even at the potential expense of failing to identify real defects.

---

**Result(1)**

<table>
<thead>
<tr>
<th>Model set</th>
<th>Execution time (min)</th>
<th>Statement coverage</th>
<th>Branch coverage</th>
<th>Total warning count</th>
<th>Using assert</th>
<th>NLL</th>
<th>print def</th>
<th>Memory leak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note</td>
<td>12</td>
<td>90.1%</td>
<td>87.9%</td>
<td>15</td>
<td>2</td>
<td>11</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>13</td>
<td>90.0%</td>
<td>86.3%</td>
<td>17</td>
<td>6</td>
<td>12</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>System &amp; api</td>
<td>23</td>
<td>73.1%</td>
<td>73.1%</td>
<td>58</td>
<td>8</td>
<td>24</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

**Result(2)**

Figure 9: Relationships between path limits, execution time and defects reported.
Continuing work

- improvement of the presentation, sorting and filtering tools
- additional classes of defects, such as race and deadlock conditions in multi-threaded code
- applying the technology to additional languages.

Limitations

- The source string in strcpy should be NULL terminated.
- The destination string in strcpy will be NULL terminated.
- The lengths of the source and destination strings will be the same length.
- The memory for the source and destination strings should not overlap.
- The length of the source and destination strings combined should not be greater than the size of available memory.

- These characteristics are typical of those missing from the modeling language.
- There are obvious limitations of expressiveness in the language as it now stands.