Checking Inside the Black Box: Regression Testing Based on Value Spectra Differences

Tao Xie       David Notkin

Dept. of Computer Science & Engineering, University of Washington, Seattle
12 Sept. 2004
ICSM 2004, Chicago
Synopsis

• Context:
  Traditional regression testing strongly focuses on black-box comparison of program outputs

• Problem:
  Behavior deviations (behavior differences) might be difficult to be propagated to observable outputs

• Goal:
  Detect and understand behavior deviations inside the black box
Approaches

Existing approaches:

• Fault propagation models [Thompson et al. 93, Voas 92]
• Structural program spectra, e.g. branch, path
  [Ball&Larus 96, Reps et al. 97, Harrold et al. 00]

Our new approach:

• Compute value spectra from a program execution
  • characterize program states of user functions
• Compare value spectra from two versions
  • detect and understand deviation propagation
Outline

- Value Spectra
- Value Spectra Comparison
- Experiment
- Conclusion
Example Program

```c
int main(int argc, char *argv[]) {
    int i, j;
    if (argc != 3) {
        printf("Wrong arguments!\n");
        return 1;
    }
    i = atoi(argv[1]);
    j = atoi(argv[2]);
    if (max(i, j) >= 0) {
        if (max(i, j) == 0) {
            printf("0\n");
        } else {
            printf("1\n");
        }
    } else {
        printf("-1\n");
    }
    return 0;
}
```
Example Program

```c
int main(int argc, char *argv[]) {
    int i, j;
    if (argc != 3) {
        printf("Wrong arguments!\n");
        return 1;
    }
    i = atoi(argv[1]);
    j = atoi(argv[2]);
    if (max(i, j) >= 0) {
        if (max(i, j) == 0) {
            printf("0");
        } else {
            printf("1");
        }
    } else {
        printf("-1");
    }
    return 0;
}
```

Program black-box input “0 1”
Program black-box output “1”
Dynamic Call Tree

main

  max

  max
Dynamic Call Tree

Main-entry state

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>&quot;0&quot;</td>
<td>&quot;1&quot;</td>
</tr>
</tbody>
</table>

main

  max

  max
Dynamic Call Tree

Main-entry state

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>&quot;0&quot;</td>
<td>&quot;1&quot;</td>
</tr>
</tbody>
</table>

Main-exit state

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>&quot;0&quot;</td>
<td>&quot;1&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>
Dynamic Call Tree

Main-entry state

main

max

Main-exit state

main(entry(3, "0", "1"), exit(3, "0", "1", 0))
Dynamic Call Tree

**Main-entry state**

```
main
    max
        max(entry(0, 1), exit(0, 1, 1))
        max
            max
                max
                    max
                        reta b 
                            0 1 1
                    max
                        reta b 
                            0 1 1
            max
                reta b 
                    0 1 1
    max
        max
            max
                reta b 
                    0 1 1
```

**Main-exit state**

```
3 "0" "1" 0
```
Value Spectra

• Value hit spectra
  • `main(entry(3, “0”, “1”), exit(3, “0”, “1”, 0))`
  • `max(entry(0, 1), exit(0, 1, 1))`

• Value count spectra
  • include additional count information

• Value trace spectra
  • include additional sequence order information
Outline

- Value Spectra
- Value Spectra Comparison
- Experiment
- Conclusion
Spectra Comparison

- Function execution comparison

- State linearization to compare the values of pointer-type variables [Xie, Marinov, Notkin ASE 04]
Understanding Deviations

entry state
≠
exit state

Deviation follower

entry state
=
exit state

Deviation container

entry state

old version

entry state

new version

exit state
≠
exit state

exit state
Understanding Deviation Propagations

• Understand where the deviations start and how they are propagated.

• Deviation root
  • a program change that triggers specific behavior deviations

• Deviation-root localization
  • two heuristics
Heuristic 1: Earliest dev follower’s preceded area

- entry state
- exit state

container or nondeviated

entry state
follower
exit state

entry state
follower
exit state

Deviation root
Deviation Follower Example

The 58th test on the 9th faulty version of tcas program

[Hutchins et al. 94]
Heuristic 2: Innermost dev container’s enclosed area

![Diagram showing the concept of innermost dev container's enclosed area with entry and exit states.](image-url)
Deviation Container Example

The 91\textsuperscript{th} test on the 9\textsuperscript{th} faulty version of tcas program

main

\begin{itemize}
\item \textbf{initialize}
\item alt_sep_test-----------------[dev container]
\item Non_Crossing_Biased_Climb
\item \hspace{1em} Inhibit_Biased_Climb
\item \hspace{2em} Own_Above_Threat
\item \hspace{3em} ALIM
\item Own_Below_Threat
\item Non_Crossing_Biased_Descend-[dev container]
\item \hspace{1em} Inhibit_Biased_Climb
\item \hspace{2em} Own_Below_Threat
\end{itemize}

[Hutchins et al. 94]
Outline

- Value Spectra
- Value Spectra Comparison
- Experiment
- Conclusion
# Experimental Subjects

[Hutchins et al. 94, Rothermel&Harrold 98]

<table>
<thead>
<tr>
<th>program</th>
<th>funcs</th>
<th>loc</th>
<th>tests</th>
<th>vers_used</th>
</tr>
</thead>
<tbody>
<tr>
<td>printtok</td>
<td>18</td>
<td>402</td>
<td>4130</td>
<td>7</td>
</tr>
<tr>
<td>printtok2</td>
<td>19</td>
<td>483</td>
<td>4115</td>
<td>10</td>
</tr>
<tr>
<td>replace</td>
<td>21</td>
<td>516</td>
<td>5542</td>
<td>12</td>
</tr>
<tr>
<td>schedule</td>
<td>18</td>
<td>299</td>
<td>2650</td>
<td>9</td>
</tr>
<tr>
<td>schedule2</td>
<td>16</td>
<td>297</td>
<td>2710</td>
<td>10</td>
</tr>
<tr>
<td>tcas</td>
<td>9</td>
<td>138</td>
<td>1608</td>
<td>9</td>
</tr>
<tr>
<td>totinfo</td>
<td>7</td>
<td>346</td>
<td>1052</td>
<td>6</td>
</tr>
</tbody>
</table>
Questions to Be Answered

• How effectively can we use the value spectra differences to expose deviations (comparing to using output differences)?

• How accurately can we use the two heuristics to locate deviation roots?
Experiment Design

• Run each test on both the original version and a faulty version (instrumented using Daikon frontend [Ernst et al. 01])
• Compute value spectra of two versions
• Compare value spectra of two versions [compare outputs of two versions]
• Locate deviation roots from spectra differences
Measurements

• Deviation exposure ratio:
  \[
  \frac{|\text{tests exhibiting spectra diffs}|}{|\text{tests covering the changed lines}|}
  \]

• Deviation-root localization ratio:
  \[
  \frac{|\text{tests succeeding in locating roots}|}{|\text{tests exhibiting spectra diffs}|}
  \]

• The higher, the better
Deviation Exposure Ratios

[Box plots showing deviation exposure ratios for different categories: output, value hit, value count, value trace.]
What We Have Learned

• When program outputs are the same for two versions, deviations can be still detected based on value spectra differences.

• Value hit spectra seem to be good enough; adding count information or sequence information does not improve much.
Deviation-Root Localization Ratios
(Value Hit Spectra)
What We Have Learned

- Identified deviation roots are accurate for most programs.

- The exceptional case is schedule2, whose state changes lie in deep parts of a linked list.
  - By default, Daikon frontend looks into state information of complex data structures with depth of three
Threats to Validity

• Representative of true practice?
  • Subject programs, faults, and tests

• Instrumentation effects that bias the results
  • Faults on tools (analysis scripts, Daikon frontend)

• Use of approximate state information
Conclusion

• Checking only black-box outputs is limited in regression testing

• Value spectra enrich the existing program spectra family

• Comparing value spectra helps detect and understand deviation propagation

• The experimental results have shown
  • Comparing value spectra is effective in detecting deviations
  • Two heuristics are effective in locating deviation roots
Questions?
### Scalability

- **Cost** = $O(|\text{vars}| \times |\text{userfuncs}| \times |\text{testsuite}|)$

<table>
<thead>
<tr>
<th>program</th>
<th>funcs</th>
<th>loc</th>
<th>tests</th>
<th>trace size/test (kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>printtok</td>
<td>18</td>
<td>402</td>
<td>4130</td>
<td>36</td>
</tr>
<tr>
<td>printtok2</td>
<td>19</td>
<td>483</td>
<td>4115</td>
<td>50</td>
</tr>
<tr>
<td>replace</td>
<td>21</td>
<td>516</td>
<td>5542</td>
<td>71</td>
</tr>
<tr>
<td>schedule</td>
<td>18</td>
<td>299</td>
<td>2650</td>
<td>982</td>
</tr>
<tr>
<td>schedule2</td>
<td>16</td>
<td>297</td>
<td>2710</td>
<td>272</td>
</tr>
<tr>
<td>tcas</td>
<td>9</td>
<td>138</td>
<td>1608</td>
<td>8</td>
</tr>
<tr>
<td>totinfo</td>
<td>7</td>
<td>346</td>
<td>1052</td>
<td>27</td>
</tr>
</tbody>
</table>

- **Execution slowdown:** (2 ~ 7) 
  - schedule2: 31
  - schedule: 48

- **Analysis time:** (7 ~ 30 ms/test) 
  - schedule2: 137
  - schedule: 201 ms/test
Related Work

• Structural program spectra [Ball & Larus 96, Reps et al. 97, Harrold et al. 00]
• GUI test oracles based on GUI states [Memon et al. 03]
• Relative debugging [Abramson et al. 96]
• Comparison checking [Jaramillo et al. 02]
• RELAY model [Thompson et al. 93]
• PIE (Propagation, Infection, and Execution) model [Voas 92]
Representation of Function Execution

- Function-entry state
  - Argument values
  - Global variable values
- Function-exit state
  - Updated argument values
  - Updated global variable values
  - Return value
- \texttt{funcname (entry(argvs), exit(argvs, ret))}
  \begin{itemize}
  \item \texttt{main(entry(3, "0", "1"), exit(3, "0", "1", 0))}
  \item \texttt{max(entry(0, 1), exit(0, 1, 1))}
  \end{itemize}