Locating Failure Causes

Andreas Zeller
Finding Causes

Infected state

Sane state

The difference causes the failure
Search in Space

Infected state

Sane state

Mixed state

Test

argc = 3
Search in Time

Failing run

Passing run

Transition from argc to a[2]

\[
\text{argvc} = 3
\]

\[
a[2] = 0
\]
Transitions

A cause transition occurs when a new variable begins to be a failure cause:

- `argc` no longer causes the failure...
- ...but `a[2]` does!

Can be narrowed down by binary search
int main(int argc, char *argv[])
{
    int *a;

    // Input array
    a = (int *)malloc((argc - 1) * sizeof(int));
    for (int i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    // Sort array
    shell_sort(a, argc);

    // Output array
    printf("Output: ");
    for (int i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
Why Transitions?

- Each failure cause in the program state is caused by some statement
- These statements are executed at cause transitions
- Cause transitions thus are statements that cause the failure!
Potential Fixes

• Each cause transition implies a fix to make the failure no longer occur – just prohibit the transition

• A cause transition is more than a potential fix – it may be “the” defect itself
Searching GCC State

Infected state

Sane state

<PLUS node>

Mixed state

Test

✔

✘

✔

?
Search in Time

Failing run

<PLUS node>

Passing run

<PLUS node>
Search in Time

Failing run

<PLUS node> \[\text{link} \rightarrow \text{fld}[0].rtx \rightarrow \text{fld}[0].rtx == \text{link}\]

Passing run
Search in Time

Failing run

<PLUS node>  
Transition from PLUS to cycle  
<Tree cycle>  

Passing run

<PLUS node>
# All GCC Transitions

<table>
<thead>
<tr>
<th>#</th>
<th>Location</th>
<th>Cause transition to variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(Start)</td>
<td>argv[3]</td>
</tr>
<tr>
<td>1</td>
<td>toplev.c:4755</td>
<td>name</td>
</tr>
<tr>
<td>2</td>
<td>toplev.c:2909</td>
<td>dump_base_name</td>
</tr>
<tr>
<td>3</td>
<td>c-lex.c:187</td>
<td>finput→_IO_buf_base</td>
</tr>
<tr>
<td>4</td>
<td>c-lex.c:1213</td>
<td>nextchar</td>
</tr>
<tr>
<td>5</td>
<td>c-lex.c:1213</td>
<td>yyss[41]</td>
</tr>
<tr>
<td>6</td>
<td>c-typeck.c:3615</td>
<td>yyss[42]</td>
</tr>
<tr>
<td>9</td>
<td>combine.c:4271</td>
<td>x→fld[0].rtx→fld[0].rtx</td>
</tr>
</tbody>
</table>
if (GET_CODE (XEXP (x, 0)) == PLUS {
    x = apply_distributive_law
    (gen_binary (PLUS, mode,
        gen_binary (MULT, mode,
            XEXP (XEXP (x, 0), 0),
            XEXP (x, 1)),
        gen_binary (MULT, mode,
            XEXP (XEXP (x, 0), 1),
            XEXP (x, 1))));

    if (GET_CODE (x) != MULT)
        return x;
}

Should be copy_rtx()
How good are we?

Evaluation using the Siemens Testsuite:

• 7 programs – most text processors
• 132 variations, each with 1 seeded defect
• Challenge: Using test runs, locate defect
• All proposed defect locators fail
  (Comparing coverage, slicing, dynamic invariants)
Close to the Defect

Predicted location

✘
Locating Defects

- NN (Renieris + Reiss, ASE 2003)
- CT (Cleve + Zeller, ICSE 2005)
- SD (Liblit et al., PLDI 2005)
- SOBER (Liu et al., TR 2005)

Results obtained from Siemens test suite; cannot be generalized.

5,542 runs

2 runs

Source code to examine

<0% failures
<10% failures
<20% failures
<30% failures
Open Issues

- Hierarchical search
- Ranking transitions
- User-side diagnosis
- Combination with statistical causality
Concepts

★ Cause transitions pinpoint *failure causes in the program code*

★ Failure-causing statements are *potential fixes* (and frequently defects, too)

★ Even more demanding, yet effective technique