Introduction to Debugging

Andreas Zeller • Saarland University

The Problem

Facts on Debugging

- Software bugs cost ~60 bln US$/yr in US
- Improvements could reduce cost by 30%
- Validation (including debugging) can easily take up to 50-75% of the development time
- When debugging, some people are three times as efficient than others
How to Debug
(Sommerville 2004)

The Process

Track the problem
Reproduce
Automate
Identify origins
Isolate
Correct
Tracking Problems

• Every problem gets entered into a problem database
• The priority determines which problem is handled next
• The product is ready when all problems are resolved
Reproduce

Randomness  Operating System
Communication  Concurrency
Interaction  Physics
Data  Debugger

Automate

// Test for host
public void testHost() {
    int noPort = -1;
    assertEquals(askigor_url.getHost(), "www.askigor.org");
    assertEquals(askigor_url.getPort(), noPort);
}

// Test for path
public void testPath() {
    assertEquals(askigor_url.getPath(), "/status.php");
}

// Test for query part
public void testQuery() {
    assertEquals(askigor_url.getQuery(), "id=sample");
}

Automate

• Every problem should be
  reproducible automatically

• Achieved via appropriate (unit) tests

• After each change, we re-run the tests
Finding Origins

1. The programmer creates a defect in the code.
2. When executed, the defect creates an infection.
3. The infection propagates.
4. The infection causes a failure.

This infection chain must be traced back – and broken.

Not every defect creates an infection – not every infection results in a failure

The Defect

Variables
A Program State

Finding Origins

1. We start with a known infection (say, at the failure)
2. We search the infection in the previous state
A Program State

Search

WHERE'S WALDO?
Focus

During our search for infection, we focus upon locations that

- are possibly wrong
  (e.g., because they were buggy before)
- are explicitly wrong
  (e.g., because they violate an assertion)

Assertions are the best way to find infections!

Finding Infections

```cpp
class Time {
  public:
    int hour();     // 0..23
    int minutes();  // 0..59
    int seconds();  // 0..60 (incl. leap seconds)
    void set_hour(int h);
  ...
}
```

Every time between 00:00:00 and 23:59:60 is valid
Finding Origins

bool Time::sane()
{
    return (0 <= hour() && hour() <= 23) &&
           (0 <= minutes() && minutes() <= 59) &&
           (0 <= seconds() && seconds() <= 60);
}

void Time::set_hour(int h)
{
    assert (sane()); // Precondition
    ...
    assert (sane()); // Postcondition
}

sane() is the invariant of a Time object:
• valid before every public method
• valid after every public method

Finding Origins

• Precondition fails = Infection before method
• Postcondition fails = Infection after method
• All assertions pass = no infection

void Time::set_hour(int h)
{
    assert (sane()); // Precondition
    ...
    assert (sane()); // Postcondition
}
Complex Invariants

```java
class RedBlackTree {
    ...
    boolean sane() {
        assert (rootHasNoParent());
        assert (rootIsBlack());
        assert (redNodesHaveOnlyBlackChildren());
        assert (equalNumberOfBlackNodesOnSubtrees());
        assert (treeIsAcyclic());
        assert (parentsAreConsistent());
        return true;
    }
}
```

Assertions

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>☘</td>
<td></td>
</tr>
</tbody>
</table>
```

Focusing

- All possible influences must be checked
- Focusing on most likely candidates
- Assertions help in finding infections fast
Isolation

- Failure causes should be narrowed down systematically
- Use observation and experiments

Scientific Method

1. Observe some aspect of the universe.
2. Invent a hypothesis that is consistent with the observation.
3. Use the hypothesis to make predictions.
4. Test the predictions by experiments or observations and modify the hypothesis.
5. Repeat 3 and 4 to refine the hypothesis.
Explicit Hypotheses

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>The execution causes $a_{01} = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prediction</td>
<td>At Line 37, $a_{01}$ should hold.</td>
</tr>
<tr>
<td>Experiment</td>
<td>Observe $a_{01}$ at Line 37.</td>
</tr>
<tr>
<td>Observation</td>
<td>$a_{01}$ holds as predicted.</td>
</tr>
<tr>
<td>Conclusion</td>
<td>Hypothesis is confirmed.</td>
</tr>
</tbody>
</table>

Keeping everything in memory is like playing mastermind blind!
Isolate

• We repeat the search for infection origins until we found the defect
• We proceed *systematically* along the scientific method
• *Explicit steps* guide the search – and make it repeatable at any time

Correction

Before correcting the defect, we must check whether the defect
• actually is an *error* and
• *causes* the failure

Only when we understood both, can we correct the defect

The Devil’s Guide to Debugging

*Find the defect by guessing:*
• Scatter debugging statements everywhere
• Try changing code until something works
• Don’t back up old versions of the code
• Don’t bother understanding what the program should do
The Devil’s Guide to Debugging

Don’t waste time understanding the problem.

- Most problems are trivial, anyway.

Use the most obvious fix.

- Just fix what you see:
  ```java
  x = compute(y)
  // compute(17) is wrong – fix it
  if (y == 17)
    x = 25.15
  ```

  Why bother going into compute()?

Successful Correction
Homework

- Does the failure no longer occur? (If it does still occur, this should come as a big surprise)
- Did the correction introduce new problems?
- Was the same mistake made elsewhere?
- Did I commit the change to version control and problem tracking?

The Process

T rack the problem
R eproduce
A utomate
F ind Origins
F ocus
I solate
C orrect

“The definitive book on debugging”
— WALTER F. TICHY
TU Karlsruhe

“Why Programs Fail”
A Guide to Systematic Debugging
ANDREAS ZELLER
46

47

48
## Failure Causes in GCC

<table>
<thead>
<tr>
<th>Location</th>
<th>Failure Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Start&gt;</td>
<td>argv[3]</td>
</tr>
<tr>
<td>toplev.c:4755</td>
<td>name</td>
</tr>
<tr>
<td>toplev.c:2909</td>
<td>dump_base_name</td>
</tr>
<tr>
<td>c-lex.c:187</td>
<td>finput → _IO_buf_base</td>
</tr>
<tr>
<td>c-lex.c:1213</td>
<td>nextchar → _IO_buf_base</td>
</tr>
<tr>
<td>c-lex.c:1213</td>
<td>yyssa[41] → _IO_buf_base</td>
</tr>
<tr>
<td>c-typeck.c:3615</td>
<td>yyssa[42] → _IO_buf_base</td>
</tr>
<tr>
<td>c-lex.c:1213</td>
<td>last_insn → fld[1].rtx → … → fld[1].rtx.code</td>
</tr>
<tr>
<td>combine.c:4271</td>
<td>x → fld[0].rtx −→ fld[0].rtx</td>
</tr>
</tbody>
</table>
Automatic Fixes!

Automatic Fixes

(a) Java Program
(b) Failing and Passing Runs
(c) Models

(d) Model Differences
(e) Fix Candidates
(f) Validated Fix

In Socket.java, line 356: > bind()
Mining Object Behavior

- **Mutators**
  - add(1)
  - remove(1)
  - change state

- **Inspectors**
  - isEmpty()
  - firstElement()
  - return state

Use static analysis to differentiate

Building Models

- After each mutator call, we extract attributes and invoke the inspectors
- Extracted states form finite state machine
Building Models

Equivalence Classes

Automatic Fixes

(a) Java Program  (b) Failing and Passing Runs  (c) Models
Automatic Fixes

(d) Model Differences

(e) Fix Candidates

(f) Validated Fix

Mina

• Multipurpose Infrastructure for Network Applications

Failing run calls unbind() although not bound Can we fix it?

Deleting Calls

• The first option to create fixes is to delete calls:

• Make calls dependent on precondition

• Or, make callees return when precondition does not hold
Inserting Calls

- The second fix option is to insert calls:
- For a violated precondition, insert calls to reach that state
- May need to traverse model for that

Fix it! Inserting Calls

- The second fix option is to insert calls:
- For a violated precondition, insert calls to reach that state
- May need to traverse model for that
Validating Fixes

All fix options must be validated:

We validate fix candidates

1. On failing test
   - Call unbind() only if bound

2. On entire test suite
   - Call bind() before unbind()

Only validated fixes remain

Pachika

Suaheli for "fix", "insert"

- Tool for automatic fixing of Java programs
- Takes a failing run and a test suite
- Produces either a validated fix – or nothing
- Available for download
For 4 out of those 7 bugs there is at least one fix. For 7 bugs, there is no fix that causes the failing test to pass. We examined the results in detail and found two causes for eight bugs. However, none of these fixes causes the failing test to pass in our experiments.

The table does not list the time model miner takes to extract models for depth 2. Tracing is turned on. The third column gives the execution time as expressed as the factor by which execution time increases when the latest version used in the experiments. The tracing overhead is always less than a second.

Table 4: Tracing overhead and execution times for all subjects:

<table>
<thead>
<tr>
<th>Bug</th>
<th>Candidate Fixes</th>
<th>Insert</th>
<th>Delete</th>
<th>Potential Fixes</th>
<th>Validated Fixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>34858</td>
<td>420</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>43033</td>
<td>219</td>
<td>65</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>51322</td>
<td>112</td>
<td>190</td>
<td>56</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>67774</td>
<td>0</td>
<td>72</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>70619</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>75129</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>87376</td>
<td>20</td>
<td>218</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>107858</td>
<td>405</td>
<td>235</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>109614</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>120474</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>121616</td>
<td>123</td>
<td>0</td>
<td>38</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>125475</td>
<td>72</td>
<td>122</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>128237</td>
<td>283</td>
<td>4</td>
<td>123</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>131933</td>
<td>0</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>152631</td>
<td>0</td>
<td>783</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>158412</td>
<td>2895</td>
<td>310</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>158624</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>173602</td>
<td>17</td>
<td>13</td>
<td>7</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 5 lists information about overhead and execution times:

- **MINA**
- **RHINO**
- **ASPECTJ**
- **JDO**
- **PACHIKA**

Overhead Size Mining

Tracing Trace File Model

acht factor

Table 6: Results of the experimental evaluation for

- **MINA**
- **RHINO**
- **ASPECTJ**
- **JDO**
- **PACHIKA**

...
Bug 51322

```java
public EclipseTypeMunger build(ClassScope cs) {
    ...
    binding = classScope.referenceContext.
        binding.resolveTypesFor(binding);
    // Fix generated by PACHIKA
    binding.constantPoolDeclaringClass().
        addDefaultAbstractMethods();
    binding.constantPoolDeclaringClass().methods();
    // Fix from source repository
    if (binding == null)
        throw new AbortCompilation();
    ResolvedMember sig = new ResolvedMember(...);
    ...
}
```

Automatic Fixing

- Adaptive fix generation
- Assessing the impact of fixes
- Leveraging contracts
- Programs that fix themselves

http://www.st.cs.uni-saarland.de/models/

Summary

Scientific Method

Finding Origins

1. The programmer creates a defect in the code.
2. When executed, the defect creates an exception.
3. The exception propagates.
4. The defective cause is fixed.

This relation also must be traced back and forth.

http://www.cs.uni-saarland.de/models/