Testing, Debugging, Program Verification

Debugging Programs

Srinivas Pinisetty[1]

10 November 2017

[1]Slides based on material from Wolfgang Aherndt,..

Student Representatives

Chalmers

- Admas Aklilu (admas)
- Rasmus Jemth (jemthr)
- Jonatan Nylund (nylundj)
- Kevin Chen Trieu (kevintr)

GU

- John Lindstrom Gidskehaug (guslinjoga)
- Daniel Beecham (gusbeeda)
- Niklasson Sebastian (gusnikse)
- Johanna Torbjornsson (gustorbjjo)

Debugging

So far: Testing

- Look for inputs that cause unexpected behaviour.
- Coverage criteria: Creating good test suits.
- Input space partitioning: Choose different/boundary inputs
- Cover as many potential problems as possible.
- Program fails, now what?

Today: Debugging

- How to systematically find source of failure.
  - Test-case to reproduce problem.
  - Finding a small failing input (if possible).
- Observing execution: Debuggers and Logging.
- Program dependencies: data- and control.

Motivation

Debugging needs to be systematic

- Bug reports may involve large inputs
- Programs may have thousands of memory locations
- Programs may pass through millions of states before failure occurs

Debugging Steps

Debugging Steps

1. Reproduce the error, understand
2. Isolate and Minimize (shrink)– Simplification
3. Eyeball the code, where could it be?– Reason backwards
4. Devise and run an experiment to test your hypothesis
5. Repeat 3,4 until you understand what is wrong
6. Fix the Bug and Verify the Fix
7. Create a Regression Test

Common Themes

- Separate relevant from irrelevant
- Being systematic: avoid repetition, ensure progress, use tools

Debugging Techniques

- Minimize (shrink)– Input simplification/problem minimization
- Observe outcome/test hypothesis– State inspection using debuggers and logging
- Tracking causes and effects — From failure to defect. Which start states cause failure?

Automatic Problem Minimisation

Observing outcome, state inspection

Tracking, reasoning backwards

Problem Simplification: Big Failing Input

This input made mozilla crash in 2002, what was the problem?

```
...
<td align=left valign=top>
<SELECT NAME="op sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION VALUE="Windows 95">Windows 95<OPTION VALUE="Windows
98">Windows 98<OPTION VALUE="Windows ME">Windows ME<OPTION VALUE="Windows 2000">Windows 2000<OPTION VALUE="Windows
NT">Windows NT<OPTION VALUE="Mac System 7">Mac System 7<OPTION VALUE="Mac System 7.5">Mac System 7.5<OPTION VALUE="Mac
System 7.6.1">Mac System 7.6.1<OPTION VALUE="Mac System 8.0">Mac System 8.0<OPTION VALUE="Mac System 8.5">Mac System
8.5<OPTION VALUE="Mac System 8.6">Mac System 8.6<OPTION VALUE="Mac System 9.x">Mac System 9.x<OPTION VALUE="MacOS X">MacOS
X<OPTION VALUE="Linux">Linux<OPTION VALUE="BSDI">BSDI<OPTION VALUE="FreeBSD">FreeBSD<OPTION VALUE="NetBSD">NetBSD<OPTION
VALUE="OpenBSD">OpenBSD<OPTION VALUE="AIX">AIX<OPTION VALUE="BeOS">BeOS<OPTION VALUE="HP-UX">HP-UX<OPTION
VALUE="IRIX">IRIX<OPTION VALUE="Neutrino">Neutrino<OPTION VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION
VALUE="OSF/1">OSF/1<OPTION VALUE="Solaris">Solaris<OPTION VALUE="SunOS">SunOS<OPTION VALUE="other">other</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION VALUE="P3">P3<OPTION VALUE="P4">P4<OPTION
VALUE="P5">P5</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="bug severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION VALUE="major">major<OPTION
VALUE="normal">normal<OPTION VALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION
VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
...
```

- Simplify failing test case into a minimal test case that still produces the failure
- How would you do this by hand?

Problem Simplification

We need a small failed test case

Simplify failing test case into a minimal test case that still produces the failure

Divide-and-Conquer

1. Cut away one half of the test input
2. Check, whether one of the halves still exhibits failure
3. Continue until minimal failing input is obtained

(Same principle as binary search!)

Problems

- Tedious: rerun tests manually
- Boring: cut-and-paste, rerun
- What, if none of the halves exhibits a failure?

Automatic Input Simplification

- Automate cut-and-paste and re-running tests
- Partition test input into n chunks

c_1	$\cdots$	c_n

- Remove one chunk at a time, re-run test on remaining pattern

c_1	$\cdots$	c_{i-1}	c_{i+1}	$\cdots$	c_n

- Increase granularity (number of chunks) when no failure occurs

Example

public static int `checkSum(`**int**`[] a)`

- is supposed to compute the checksum of an integer array
- gives wrong result, whenever `a` contains two identical consecutive numbers, but we don't know that yet
- we have a failed test case, e.g., from protocol transmission:

  {`1,3,5,3,9,17,44,3,6,1,1,0,44,1,44,0`}

Input Simplification ($n =$ number of chunks)

	1	3	5	3	9	17	44	3	6	1	1	0	44	1	44	0	✘
n=2																	
	1	3	5	3	9	17	44	3									✔
									6	1	1	0	44	1	44	0	✘
									6	1	1	0					✘
									6	1							✔
											1	0					✔
n=4	increase granularity																
									6	1	1						✘
n=3	adjust granularity to input size																
									6	1							✔
										1	1						✘

The ddMin algorithm

- Let c be a failing input configuration (sequence of individual inputs).
- test(c) runs a test on c with possible outcome `PASS` or `FAIL`.
- `n` is the number of chunks to split c into (initially n = 2). We will remove one chunk at the time, and test the remaining input.

ddMin(c, n) =

1. If $|c| = 1$ **return** c

Otherwise, systematically remove one chunk c_i at the time.
Test the remaining input $c \backslash c_i$:

2. If there exist some c_i such that $test(c \backslash c_i) = FAIL$[2]
   return ddMin($c \backslash c_i$, max(n-1, 2))
3. Else, if $n < |c|$ **return** ddMin(c, min(2n, $|c|$))
4. Else, (can't split into smaller chunks) **return** c

[2] In our example, we start by removing the last chunk. But the order does not actually matter for the algorithm, it's an implementational choice.

Mini Quiz: ddMin

ddMin(c, n) =

1. If $|c| = 1$ **return** c

Otherwise, systematically remove one chunk c_i at the time.
Test the remaining input $c \backslash c_i$:

2. If there exist some c_i such that $test(c \backslash c_i) = FAIL$
   return
   ddMin($c \backslash c_i$, max(n-1, 2))
3. Else, if $n < |c|$
   return ddMin(c, min(2n, $|c|$))
4. Else, (can't split into smaller chunks) **return** c

- Let $test(c)$ return `FAIL` whenever c contains two or more occurrences of the letter 'X'.
- Apply the ddMin algorithm to minimise the failing input array [X, Z, Z, X]
- Write down each step of the algorithm, and the values of n (number of chunks).
  Initially, n is 2.

Mini Quiz: Solution

Initial failing input: [X, Z, Z, X]
Initial n = 2, split into two chunks:

- [X, Z]⇒ `PASS` (remove 2nd chunk)
- [Z, X]⇒ `PASS` (remove 1st chunk)

Update n = 4 (see step 3), split into four chunks:

- [X, Z, Z]⇒ `PASS` (remove 4th chunk)
- [X, Z, X]⇒ `FAIL` (remove 3rd chunk)

Update n = 3 (see step 2), split into three chunks:

- [X, Z]⇒ `PASS` (remove 3rd chunk)
- [X, X]⇒ `FAIL` (remove 2nd chunk)

Update n = 2 (see step 2), split into two chunks:

- [X]⇒ `PASS` (remove 2nd chunk)
- [X]⇒ `PASS` (remove 1st chunk)

No further splits possible, minimal failing input is [X, X]

Minimal Failure Configuration

Consequences of Minimisation

- Input small enough for observing, tracking, locating (next topics)
- Minimal input often provides important hint for source of defect.

Implementation

- For details on implementation of minimisation algorithm, see Zeller ch. 5 (in particular 5.4-5.5).
- Play with the Java implementation: `DD.java` and `Dubbel.java`
  (+ exercise session!).

Automatic Problem Minimisation

Observing outcome, state inspection

Tracking, reasoning backwards

Observing outcome, state inspection

Observation of intermediate state not part of functionality!!

How can we observe the computations in a program run?

- Simple logging: print statements
- Advanced logging: configureable what is printed based on level (OFF $<$ FINE ... $<$ INFO $<$ WARNING $<$ SEVERE)
- Debugging tools: such as the eclipse debugger

The Quick & Dirty Approach: Print Logging

Println Debugging

Manually add print statements at code locations to be observed

```
System.out.println("size␣=␣"+ size);
```

- ✔ Simple and easy
- ✔ No tools or infrastructure needed, works on any platform
- ✘ Code cluttering
- ✘ Output cluttering
- ✘ Performance penalty, possibly changed behaviour (real time apps)
- ✘ Buffered output lost on crash
- ✘ Source code access required, recompilation necessary

Basic Logging in Java

See:

https://docs.oracle.com/javase/7/docs/api/java/util/logging/Logger.html

- Each class can have its own `logger` object
- Each logger has level:
  `OFF` < `FINE`... < `INFO` < `WARNING` < `SEVERE`
- Setting the level controls which messages gets written to log.

Quick Demo: `Dubbel.java`

Evaluation of Logging Frameworks

- ✔ Output cluttering can be mastered
- ✔ Small performance overhead
- ✔ Exceptions are loggable
- ✔ Log complete up to crash
- ✔ Instrumented source code reconfigurable w/o recompilation
  - See class `java.util.logging.LogManager`.
  - Logging configurations from file.
- ✘ Code cluttering — don't try to log everything!

Code cluttering avoidable with Debuggers

- post-mortem vs. interactive debugging
- Note: Not always possible to use debugger.
  - E.g. bugs in complex, large systems with timing issues.

Using Debuggers

Assume we have found a small failing test case and identified the faulty component.

Basic Functionality of a Debugger

- **Execution Control**: Stop execution at specific locations: breakpoints
- **Interpretation**: Step-wise execution of code
- **State Inspection**: Observe values of variables and stack
- **State Change**: Change state of stopped program

- We use the built-in GUI-based debugger of the ECLIPSE framework
  - You will get a chance to get practical experience with the Eclipse debugger in the exercise session next week.
  - Feel free to experiment with other debuggers!

Automatic Problem Minimisation

Observing outcome, state inspection

Tracking, reasoning backwards

Tracking Causes and Effects

Determine defect that is origin of failure

Fundamental problem

Programs executed forward, but need to reason backward from failure

Reasoning Backwards

How do we know which statements influences other statements?

Running Example: Binary Search

```
public static int search( int array[], int
 target ) {

int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
mid = (low + high)/2;
if ( target < array[ mid ] ) {
high = mid - 1;
} else if ( target > array[ mid ] ) {
low = mid + 1;
} else {
return mid;
}
}
return -1;
}
```

Exercise and Demo: Find the bug!

Example

```
public static int search( int array[], int
 target ) {

  int low = 0;
  int high = array.length;
  int mid;
  while ( low <= high ) {
    mid = (low + high)/2;
    if ( target < array[ mid ] ) {
        high = mid - 1;
    } else if ( target > array[ mid ] ) {
        low = mid + 1;
    } else {
        return mid;
    }
  }
  return -1;
}
```

Effects of Statements

Fundamental ways how statements may affect each other

Write	Change the program state Assign a new value to a variable read by another statement
Control	Change the program counter Determine which statement is executed next

Statement Dependencies

Definition (Data Dependency)

Statement `B` is data dependent on statement `A` iff

1. `A` writes to a variable `v` that is read by `B` and
2. There is at least one execution path between `A` and `B` in which `v` is not assigned another value.

"The outcome of `A` can directly influence a variable read in `B`"

Definition (Control Dependency)

Statement `B` is control dependent on statement `A` iff

- `B`'s execution is potentially controlled by `A`

"The outcome of `A` can influence whether `B` is executed"

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
   mid = (low + high)/2;
  if ( target < array[mid] ) {
      high = mid - 1;
  } else if ( target > array[ mid ] ) {
      low = mid + 1;
  } else {
      return mid;
  }
}
return -1;
```

statement is data-dependent on this statement

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ) {
  mid = (low + high)/2;
   if ( target < array[mid] ) {
     high = mid - 1;
  } else if ( target > array[ mid ] ) {
     low = mid + 1;
  } else {
     return mid;
  }
}
return -1;
```

statement is control-dependent on the while statement

Direct Backwards Dependent

Statement B is (Directly) backwards dependent on A if either or both:

- B is control-dependent on A
- B is data-dependent on A

Example

```
int low = 0;
int high = array.length;
int mid;
while ( low <= high ){
        mid = (low + high)/2;
        if ( target < array[ mid ] ){
                high = mid - 1;
        }
        else if ( target > array[ mid ] ) {
                low = mid + 1;
        }
        else {
        return mid;
        }
}
return -1;
```

Backwards Dependent

Statement B is backwards dependent on A if B is directly backwards dependent on A in one or more steps

Tracking Down Infections

Systematic localization of defects

Let $\mathcal{I}$ be a set of infected locations (variable+program counter)
Let L be the current location in a failed execution path

1. Let L be infected location reported by failure and set $\mathcal{I} := \{L\}$
2. Compute statements $\mathcal{S}$ that potentially contain origin of defect:
   one level of backward dependency from L in execution path
3. Inspect locations $L_1, \ldots, L_n$ written to in $\mathcal{S}$:
   check if they are infected, let $\mathcal{M} \subseteq \{L_1, \ldots, L_n\}$ be infected ones
4. If one of the L_i is infected, i.e., $\mathcal{M} \neq \emptyset$:
   4.1 Let $\mathcal{I} := (\mathcal{I} \backslash \{L\}) \cup \mathcal{M}$ (replace L with the new candidates in $\mathcal{M}$)
   4.2 Let new current location L be any location from $\mathcal{I}$
   4.3 Goto 2.
5. L does not depend on any other location (must be the infection site!)

After Fixing the Defect: Testing!

- Failures that exhibited a defect become new test cases after the fix
  - used for regression testing
- During/after fixing the bug use existing unit test cases to
  - test a suspected method in isolation
  - make sure that your bug fix did not introduce new bugs
  - exclude wrong hypotheses about the defect

Summary

In this lecture, we have learned:

- How one can find a minimal failing test-case, and why this is helpful for debugging.
- Logging and debuggers.
- How to go about finding a bug systematically.
- What it means for a program statement to be
  - control dependent and data dependent.
  - How to use these concepts to help locating bugs.

What Next?

Unsolved problems

1. When does a program have no more bugs?
   How to prove correctness without executing ∞ many paths?

Remaining topics in this course that give some answers

1. Formal Specification
2. Verifying Program Correctness