Delta Debugging

CS169
Debugging

• Debugging is invoked when testing finds a bug
  – Hard task, but good tools can help a lot

• Hard problems in debugging
  – Reproduce the bug reliably
  – Find a minimal test case that reproduces the bug
    • Minimal = Each element is relevant
  – Locate the fault in the source code
  – Fix the fault
Motivation - Example

• The Firefox open-source web browser project receives several dozens bug reports a day.
• Each bug report has to be simplified
  – Eliminate all details irrelevant to producing the failure
    • To facilitate debugging
    • To make sure it does not replicate a similar bug report
• At some point Firefox listed more than 800 open bug reports
  – These involved complex HTML pages
  – Mozilla engineers were overwhelmed with work
  – They created the Mozilla BugAThon: a call for volunteers to process bug reports
  – Most common task was to simplify the reproducing scenario
Motivation

• Simplifying meant turning bug reports into minimal test cases
  – where every part of the input would be relevant to reproducing the failure

• What we want is the simplest HTML page that still produces the fault.
How do we go from this …

```html
<SELECT NAME="op sys" MULTIPLE SIZE=7>
  <OPTION VALUE="All">All</OPTION>
  <OPTION VALUE="Windows 3.1">Windows 3.1</OPTION>
  <OPTION VALUE="Windows 95">Windows 95</OPTION>
  <OPTION VALUE="Windows 98">Windows 98</OPTION>
  <OPTION VALUE="Windows ME">Windows ME</OPTION>
  <OPTION VALUE="Windows 2000">Windows 2000</OPTION>
  <OPTION VALUE="Windows NT">Windows NT</OPTION>
  <OPTION VALUE="Mac System 7">Mac System 7</OPTION>
  <OPTION VALUE="Mac System 7.5">Mac System 7.5</OPTION>
  <OPTION VALUE="Mac System 7.6.1">Mac System 7.6.1</OPTION>
  <OPTION VALUE="Mac System 8">Mac System 8.0</OPTION>
  <OPTION VALUE="Mac System 8.5">Mac System 8.5</OPTION>
  <OPTION VALUE="Mac System 8.6">Mac System 8.6</OPTION>
  <OPTION VALUE="Mac System 9.x">Mac System 9.x</OPTION>
  <OPTION VALUE="MacOS X">MacOS X</OPTION>
  <OPTION VALUE="Linux">Linux</OPTION>
  <OPTION VALUE="FreeBSD">FreeBSD</OPTION>
  <OPTION VALUE="NetBSD">NetBSD</OPTION>
  <OPTION VALUE="OpenBSD">OpenBSD</OPTION>
  <OPTION VALUE="AIX">AIX</OPTION>
  <OPTION VALUE="BeOS">BeOS</OPTION>
  <OPTION VALUE="HP-UX">HP-UX</OPTION>
  <OPTION VALUE="IRIX">IRIX</OPTION>
  <OPTION VALUE="Neutrino">Neutrino</OPTION>
  <OPTION VALUE="OpenVMS">OpenVMS</OPTION>
  <OPTION VALUE="OS/2">OS/2</OPTION>
  <OPTION VALUE="OSF/1">OSF/1</OPTION>
  <OPTION VALUE="Solaris">Solaris</OPTION>
  <OPTION VALUE="SunOS">SunOS</OPTION>
  <OPTION VALUE="other">other</OPTION>
</SELECT>

<SELECT NAME="priority" MULTIPLE SIZE=7>
  <OPTION VALUE="--">--</OPTION>
  <OPTION VALUE="P1">P1</OPTION>
  <OPTION VALUE="P2">P2</OPTION>
  <OPTION VALUE="P3">P3</OPTION>
  <OPTION VALUE="P4">P4</OPTION>
  <OPTION VALUE="P5">P5</OPTION>
</SELECT>

<SELECT NAME="bug severity" MULTIPLE SIZE=7>
  <OPTION VALUE="blocker">blocker</OPTION>
  <OPTION VALUE="critical">critical</OPTION>
  <OPTION VALUE="major">major</OPTION>
  <OPTION VALUE="normal">normal</OPTION>
  <OPTION VALUE="minor">minor</OPTION>
  <OPTION VALUE="trivial">trivial</OPTION>
  <OPTION VALUE="enhancement">enhancement</OPTION>
</SELECT>
```
The crash is when `<SELECT>` does not have closing tag
Why Simplify?

• Ease of communication. A simplified test case communicates better what the issue is.

• Easier debugging. Smaller test cases result in smaller states and shorter executions.

• More likely to be reusable as regression tests as the code evolves.

• Identify duplicates. Simplified test cases subsume several duplicates.
Delta Debugging Simplified Usage Scenario

• A program exhibits a failure on an input
  – E.g., input = a text file for a compiler
  – E.g., input = a set of numbers for sorting
  – E.g., input = a set of rows in a DB for some processing
  – E.g., input = a set of configuration parameters
  – E.g., input = a set of HTTP requests for a web service
  – E.g., input = a sequence of API calls to a module

• Find the smallest subset of the input for which the program still has the same failure
Minimizing Reproducing Input

• Important:
  – Some subsets of the Input may not generate **FAIL**
    • Code may produce some other result
    • Code may give an error message
      – e.g., compiler syntax error if we give it an ill-formed subset of the source file
A Generic Algorithm

- How do people solve these problems?

- Binary search
  - Cut the input in half
  - Try to reproduce the bug
  - Iterate
Delta Debugging: Assumptions

• There is a set of input elements $I$
  – If we use the entire $I$ we get a failure (result FAIL)
  – Need to find the minimal reproducing input (MRI) that results in failure (MRI subset of $I$)

• Assumptions
  • A1: Every subset of the input that contains the MRI will result in FAIL (monotonicity)
Delta Debugging: Monotonicity

• **Examples of monotonicity:**
  – Compiler fails as long as “1e-100” appears in the code
    • “1e-100” is the MRI
    • Still fail if other input elements appear in addition to MRI
  – Sorting fails as long as the input contains one negative and one positive number
    • The MRI is [-1, 1]
    • Still fails if other input elements appear in addition to MRI
  – One HTTP request crashes the web service
    • The MRI is that request
    • Still fails if other requests are made in addition to MRI

• **Example of non-monotonicity?**
Delta Debugging: Monotonicity

• **Examples of non-monotonicity:**
  – Compiler fails as long as “1e-100” appears in the code, except if “1e+100” also appears
    • “1e-100” is the MRI
    • But “1e-100 * 1e+100” contains the MRI and does not fail

  – Sorting fails if the input contains -1 and there are an even number of elements
    • The MRI is [-1, 1]
    • But [-1, 0, 1] contains the MRI and does not fail
• One more simplifying assumption (for now)

  • A2: There exists one input element that causes the failure by itself
    – MRI is of size 1
    – E.g., one of the HTTP requests crashes the server
    – E.g., sorting fails if 0 is part of the input
Binary Search

• Proceed by binary search.

• If input \( I \) results in failure

• Try the first half, and see if it results in FAIL
  – If yes, continue search in first half
  – If no, continue search in second half
**Version 1: Example**

- Assume $I = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$
  - The bug is due to input element 7

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>FAIL</td>
</tr>
<tr>
<td>1 2 3 4</td>
<td>NO</td>
</tr>
<tr>
<td>5 6 7 8</td>
<td>FAIL</td>
</tr>
<tr>
<td>5 6</td>
<td>NO</td>
</tr>
<tr>
<td>7 8</td>
<td>FAIL</td>
</tr>
<tr>
<td>7</td>
<td>FAIL</td>
</tr>
</tbody>
</table>
/* invariant: P fails with inputs $i_1,...,i_n$ i.e. $\{i_1,...,i_n\} \rightarrow$ FAIL
A$_2$: There is one $i_k$ that makes P fail
Find: $i_k$ */

DD($\{i_1,...,i_n\}$) =
  if $n = 1$ return $\{i_1\}$
  let $I_1 = \{i_1 \ldots i_{n/2}\}$
  let $I_2 = \{i_{n/2+1} \ldots i_n\}$
  if $I_1 \rightarrow$ FAIL then
    return DD($\{i_1 \ldots i_{n/2}\}$)
  else
    assert $I_2 \rightarrow$ FAIL (because of our assumptions)
    return DD($\{i_{n/2+1} \ldots i_n\}$)
Demonstration of Delta Debugging

- I once had to build a specialized code obfuscator/minifier for Lua
  - Rename identifiers (variables, functions) with short meaningless names
  - Remove spaces
  - Resulting code should have same behavior

```lua
-- Clear code
local function incomplete (msg)
  if msg then
    local ender = LUA_QL("<eof>")
    if string_sub(msg, -#ender) == ender then
      return true
    end
  end
  return false
end

-- Obfuscated and minified code
local function jn(O0) if O0 then local AL=PY("<eof>") if bW(O0,-#AL)==AL then return true end end return false end
```
Tricky Obfuscation Bugs

- Common bug: The obfuscator renames reserved names, or library function names (io.write)
  - The code starts to misbehave
  - Debugging obfuscated code is a major pain!

```lua
lua: build/myluaint.lua:1: attempt to call upvalue 'gf' (a nil value)
stack traceback:
  build/myluaint.lua:1: in function 'e6'
  build/myluaint.lua:1: in function 'nM'
  build/myluaint.lua:1: in function 'IU'
  build/myluaint.lua:1: in main chunk
  [C]: ?
```

- Now do you track such bugs?
Demonstration of Delta Debugging (Cont.)

• Process:

1. Find wrong behavior in obfuscated code
   • Ideally in automated tests
   • You check that this behavior is due to obfuscation

2. You collect the set of all identifiers in the program
   • Assumption: one of the identifiers when obfuscated -> FAIL

3. You change obfuscator to be able to restrict the range of identifiers to be obfuscated
   • Now you can do selective obfuscation

4. You use delta debugging to narrow down the set of obfuscated identifiers that cause the problem
Demo now …

• See file linked in Lecture Materials
Delta Debugging - Version 1: Comments

• This is just binary search!

• A very sensible algorithm to try first
  – By hand, or automated

• But it is not enough in most cases in practice
Delta Debugging - Version 1: Comments

• Let’s look at the assumptions we used so far
  – A2: the MRI is of size 1
  – If \((I_1 + I_2) -> FAIL\) then
    • Either \(I_1 -> FAIL\) and \(I_2 -> NO\)
    • Or \(I_1 -> NO\) and \(I_2 -> FAIL\)

• It becomes interesting when the MRI is of size larger than 1
Delta Debugging – MRI of size more than 1

• A sorting function that fails if the input contains both negative and positive values
  – MRI is [-1, 1] (size at least 2)
• If you make two HTTP requests to delete shopping cart, the second one crashes
• A compiler crash typically cannot be reproduced with a 1-character source file
  – MRI has more than 1 character
• We need to drop the simplifying assumption that the MRI is of size 1
  – But we still need monotonicity
Scenarios

• Try binary search:
  – Partition input I into halves I₁ and I₂
  – If I₁ -> FAIL, recurse with I₁
  – Otherwise, if I₂ -> FAIL, recurse with I₂

• Notes:
  – The only other possibility is
    I₁ -> NO and I₂ -> NO
  – What happens in this case?
Interference

• By monotonicity, if $I_1 \to \text{NO}$ and $I_2 \to \text{NO}$ then
  • No subset of $I_1$ or $I_2$ causes $\text{FAIL}$ by itself
  • Yet, $I_1 + I_2 \to \text{FAIL}$

  – So, the $\text{FAIL}$ must be due to a combination of elements from $I_1$ and $I_2$

  – This is called interference (the two halves interfere)

  – Addressing interference is the major innovation in Delta Debugging, and the main enabler in practice
**Interference Example**

Consider 8 input elements, of which 3, 5 and 7 cause the **FAIL**, but only when applied together.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td><strong>FAIL</strong></td>
</tr>
<tr>
<td>1 2 3 4</td>
<td><strong>NO</strong></td>
</tr>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td><strong>NO - interference</strong></td>
</tr>
<tr>
<td>1 2 3 4 5 6</td>
<td><strong>NO</strong></td>
</tr>
<tr>
<td>1 2 3 4 7 8</td>
<td><strong>NO - interference</strong></td>
</tr>
<tr>
<td>1 2 3 4 5 6</td>
<td><strong>FAIL</strong></td>
</tr>
<tr>
<td>1 2 3 4</td>
<td><strong>FAIL</strong></td>
</tr>
<tr>
<td>1 2 5 7</td>
<td><strong>FAIL</strong></td>
</tr>
<tr>
<td>3 4 5 7</td>
<td><strong>FAIL</strong></td>
</tr>
<tr>
<td>3 5 7</td>
<td><strong>FAIL</strong></td>
</tr>
</tbody>
</table>

**IDEA:** We keep all of 1 2 3 4 and search for the minimal subset of 5 6 7 8 that, in combination with 1 2 3 4 gives failure.

**IDEA:** We keep all of 5 6 and 7 8 and search for the minimal subset of 3 4 5 6 that, in combination with 1 2 3 4 5 6 gives failure.
Handling Interference

• **Review:** The cute trick:
  – Consider $I_1 + I_2 \rightarrow \text{FAIL}$ but $I_1 \rightarrow \text{NO}$ and $I_2 \rightarrow \text{NO}$
    • Find minimal $M_2$ in $I_2$ such that $(I_1 + M_2) \rightarrow \text{FAIL}$
    • All elements in $M_2$ are necessary for $\text{FAIL}$ (along with some from $I_1$)
  – Consider now $I_1 + M_2 \rightarrow \text{FAIL}$
    • Find minimal $M_1$ in $I_1$ such that $(M_1 + M_2) \rightarrow \text{FAIL}$
    • All elements in $M_1$ are necessary for $\text{FAIL}$ (along with all of $M_2$)
  – Then all elements in $M_1 + M_2$ are necessary for $\text{FAIL}$
  – This is also minimal
Another Interference Example

Consider crash due to `<select>` in input `<select size=5>`, with each character being an input element.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;select size=5&gt;</code></td>
<td>FAIL</td>
</tr>
<tr>
<td><code>&lt;select size=5&gt;</code></td>
<td>NO</td>
</tr>
<tr>
<td><code>&lt;select size=5&gt;</code></td>
<td>NO - interf.</td>
</tr>
<tr>
<td><code>&lt;select e=5&gt;</code></td>
<td>NO</td>
</tr>
<tr>
<td><code>&lt;select e=</code></td>
<td>FAIL</td>
</tr>
<tr>
<td><code>&lt;select 5&gt;</code></td>
<td>FAIL</td>
</tr>
<tr>
<td><code>&lt;select 5&gt;</code></td>
<td>NO</td>
</tr>
<tr>
<td><code>&lt;select &gt;</code></td>
<td>FAIL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;sel ec t&gt;</code></td>
<td>NO</td>
</tr>
<tr>
<td><code>&lt;select t&gt;</code></td>
<td>FAIL</td>
</tr>
<tr>
<td><code>&lt;select t&gt;</code></td>
<td>NO</td>
</tr>
<tr>
<td><code>&lt;select ct&gt;</code></td>
<td>NO – interf</td>
</tr>
<tr>
<td><code>&lt;select ct&gt;</code></td>
<td>FAIL</td>
</tr>
<tr>
<td>......</td>
<td>......</td>
</tr>
</tbody>
</table>
Algorithm

/* I along with all elements $i_1, \ldots, i_n$ fails.
   I by itself does not fail.
   Find smallest subset of $i_1, \ldots, i_n$ that along with I still fails */

DD(I, \{i_1, \ldots, i_n\}) =

  if $n = 1$ return \{i_1\}
  let $I_1 = I + \{i_1 \ldots i_{n/2}\}$
  let $I_2 = I + \{i_{n/2 + 1} \ldots i_n\}$

if $I_1 \rightarrow$ FAIL then return DD($I$, \{i_1 \ldots i_{n/2}\})
else if $I_2 \rightarrow$ FAIL then return DD($I$, \{i_{n/2 + 1} \ldots i_n\})
else
  $M_2 = DD(I_1, \{i_{n/2 + 1} \ldots i_n\})$
  $M_1 = DD(M_2, \{i_1 \ldots i_{n/2}\})$

return $M_1 + M_2$

(Homework: There is a bug in this algorithm. Check the invariant.)
Complexity

• If a single input element induces the failure, then logarithmic
  – Why?

• Otherwise, linear in the worst case
  – Assumes constant time per invocation
  – In reality it is better than linear
The GNU C Compiler

- This input program (<code>bug.c</code>) causes GCC 2.95.2 to crash when all optimizations are enabled
- We would like to minimize this input in order to debug GCC
- For the sake of simplicity, we model each character in input as an element
  - The input is the sequence of characters

```c
#define SIZE 20
double mult(double z[], int n)
{
    int i, j;
    i = 0;
    for (j = 0; j < n; j++) {
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}

void copy(double to[], double from[], int count)
{
    int n = (count + 7) / 8;
    switch (count % 8) do {
        case 0: *to++ = *from++;
        case 7: *to++ = *from++;
        case 6: *to++ = *from++;
        case 5: *to++ = *from++;
        case 4: *to++ = *from++;
        case 3: *to++ = *from++;
        case 2: *to++ = *from++;
        case 1: *to++ = *from++;
    } while (--n > 0);
    return mult(to, 2);
}

int main(int argc, char *argv[])
{
    double x[SIZE], y[SIZE];
    double *px = x;
    while (px < x + SIZE)
    {
        *px++ = (px - x) * (SIZE + 1.0);
    }
    return copy(y, x, SIZE);
}
```
The GNU C Compiler

• The test procedure would
  – create the appropriate subset of bug.c
  – feed it to GCC
  – Return FAIL iff GCC had crashed, and NO otherwise
The GNU C Compiler

• The minimized code is

```c
int t(double z[], int n)
{int i, j; for (; ; )
{i = i + j + 1; z[i] = z[i] * (z[0] + 0);}
return z[n];}
```

• The test case is 1-minimal
  – No single character can be removed without removing the failure
  – Even every superfluous whitespace has been removed
  – The function name has shrunk from `mult` to a single `t`
  – This program actually has a semantic error (infinite loop), but GCC still isn't supposed to crash

• So where could the bug be?
  – We already know it is related to optimization
  – If we remove the `-O` option to turn off optimization, the failure disappears
The GNU C Compiler

• The GCC documentation lists 31 options to control optimization on Linux x86

  -ffloat-store -fno-default-inline -fno-defer-pop
  -fforce-mem -fforce-addr -fomit-frame-pointer
  -fno-inline -finline-functions -fkeep-inline-functions
  -fkeep-static-consts -fno-function-cse -ffast-math
  -fstrength-reduce -fthread-jumps -fcse-follow-jumps
  -fcse-skip-blocks -frerun-cse-after-loop -frerun-loop-opt
  -fgcse -fexpensive-optimizations -fschedule-insns
  -fschedule-insns2 -ffunction-sections -fddata-sections
  -fcaller-saves -funroll-loops -funroll-all-loops
  -fmove-all-movables -freduce-all-givs -fno-peeaphole
  -fstrict-aliasing

• It turns out that applying all of these options causes the failure to disappear
  – Some option(s) prevent the failure
The GNU C Compiler

• We can use test case minimization in order to find the preventing option(s)
  – Each input stands for a GCC optimization option
  – Having all optim. applied means to run GCC with no option (failing)
  – Having no optim. applied means to run GCC with all options (passing)

• After seven tests, the single option `-ffast-math` is found to prevent the failure
  – Unfortunately, it is a bad candidate for a workaround because it may alter the semantics of the program

  – Thus, we remove `-ffast-math` from the list of options and make another run

  – Again after seven tests, it turn out that `-fforce-addr` also prevents the failure

  – Further examination shows that no other option prevents the failure
The GNU C Compiler

• So, this is what we can send to the GCC maintainers
  – The minimal test case
  – “The failure only occurs with optimization”
  – “-ffast-math and -fforce-addr prevent the failure”
Another Application

- Yesterday, my program worked. Today, it does not. Why?
  - The release 4.17 of GDB changed 178,000 lines from 4.16
  - it no longer integrated properly with DDD (a graphical front-end)
  - How to isolate the change that caused the failure.
Results

• **Isolates problematic change in gdb**
  – After lots of work (by machine)
    • 178,000 lines changed, grouped into 8700 groups
    • Can do 230 builds/tests in 24 hours
    • Would take 37 days to try 8700 groups individually
    • The algorithm did this in 470 tests (48 hours)

  – Doing this by hand would be a nightmare
The Importance of Input Elements

- Essential for delta debugging is the notion of input element
  - We must be able to express the difference between the good and bad examples as a set of input elements

- The notion of input element matters:
  - Poor notion of elements $\Rightarrow$ many unresolved tests
  - Performance goes from linear (or sub-linear) to quadratic

- But notion of input element is semantic
  - Not easy to capture in a general way in a tool
  - You must think how to phrase the problem for delta debugging
Opinion

• Delta Debugging is a technique, not a tool

• Be prepared:
  – Must be reimplemented for each significant system
  – To exploit knowledge of input elements

• Good News:
  – Relatively simple algorithm, significant payoff
  – It’s worth reimplementing when you suspect it works