CS510 Software Engineering

Delta Debugging and Statistical Debugging

Asst. Prof. Mathias Payer

Department of Computer Science
Purdue University

TA: Scott A. Carr
Slides inspired by Xiangyu Zhang

http://nebelwelt.net/teaching/15-CS510-SE

Spring 2015

Table of Contents

Delta Debugging

- In the last millennium, the bug database for the Mozilla browser listed more than 370 open bugs.
- Each bug in the database describes a scenario which causes the software to fail (crash).
- These scenarios are often overly verbose: (i) they are not simplified, (ii) they may contain a lot of irrelevant information, (iii) bugs could be equivalent.
- Mozilla developers asked the community for help: process bug reports and simplify them by turning them into a minimal test case that still produces the failure.

Example Bug Report

```
<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>
```

Example Bug Report

- This crashed an older version of Mozilla.
- It's almost impossible to filter out the cause of the bug just by looking at the test case.
- The execution trace does not help much either.
- Can we simplify the test case and still trigger the same bug?
- A more desirable bug report would be: Printing an HTML file that consists of <SELECT> causes Mozilla to crash.
- Can we automate test case reduction?

Delta Debugging

- Minimize test-cases: if you change any thing in the test case the bug is no longer triggered[1].
- Let's use a smaller bug report as running example:
  `<SELECT NAME="priority" MULTIPLE SIZE=7>`
- How can we simplify this input?
- Idea: remove parts of the input and see if the program still crashes (i.e., minimize the test case).
- For the above example assume that we remove characters of the input file and start the program with this new test case.

[1] Andreas Zeller and Ralf Hildebrandt, Simplifying and Isolating Failure-Inducing Input, IEEE Trans. SE, 2002.

Simplifying and Isolating Failure-Inducing Input

1: <SELECT NAME="priority" MULTIPLE SIZE=7>: F
2: <SELECT NAME="priority" MULTIPLE SIZE=7>: P
3: <SELECT NAME="priority" MULTIPLE SIZE=7>: P
4: <SELECT NAME="priority" MULTIPLE SIZE=7>: P
5: <SELECT NAME="priority" MULTIPLE SIZE=7>: F
6: <SELECT NAME="priority" MULTIPLE SIZE=7>: F
7: <SELECT NAME="priority" MULTIPLE SIZE=7>: P
8: <SELECT NAME="priority" MULTIPLE SIZE=7>: P
9: <SELECT NAME="priority" MULTIPLE SIZE=7>: P
10: <SELECT NAME="priority" MULTIPLE SIZE=7>: F

...

26: <SELECT NAME="priority" MULTIPLE SIZE=7>: F

Delta Debugging: dd_{min}

- After 26 tries our example is reduced to <SELECT>.
- After 57 tests the original example above is reduced to <SELECT>.
- Delta Debugging automates this approach of repeated trials to reduce the input.

Delta Debugging Algorithm

Let *test* and c_x be given so that $test(0) = P \wedge test(c_x) = F$. The goal is to find $c_x^{'} = ddmin(c_x)$ such that $c_x^{'} \subseteq c_x$, $test(c_x^{'}) = F$ and $c_x^{'}$ is 1-minimal. The *minimizing Delta Debugging algorithm ddmin(c)* is:

$$ddmin(c_x) = ddmin_2(c_x, 2) \text{ where}$$

$$ddmin_2(c_x^{'}, n) = \begin{cases} ddmin_2(\triangle_i, 2) & \text{if } \exists i \in \{1, ..., n\} \cdot test(\triangle_i) = F \\ & \text{(reduce testset)} \\ ddmin_2(\nabla_i, max(n-1, 2)) & \text{else if } \exists i \in \{1, ..., n\} \cdot test(\nabla_i) = F \\ & \text{(reduce to complement)} \\ ddmin_2(c_x^{'}, min(|c_x^{'}|, 2n)) & \text{else if } n < |c_x^{'}| \\ & \text{(increase granularity)} \\ c_x^{'}, & \text{otherwise: done} \end{cases}$$

Delta Debugging Example

Step	Test case									*test*	
1	$\Delta_1 = \nabla_2$	1	2	3	4	.	.	.	.	?	Testing Δ_1, Δ_2
2	$\Delta_2 = \nabla_1$	.	.	.	.	5	6	7	8	?	$\Rightarrow$ Increase granularity
3	Δ_1	1	2	.	.	.	.	.	.	?	Testing $\Delta_1, \ldots, \Delta_4$
4	Δ_2	.	.	3	4	.	.	.	.	✔	
5	Δ_3	.	.	.	.	5	6	.	.	✔	
6	Δ_4	.	.	.	.	.	.	7	8	?	
7	∇_1	.	.	3	4	5	6	7	8	?	Testing complements
8	∇_2	1	2	.	.	5	6	7	8	✘	$\Rightarrow$ Reduce to $c'_\times = \nabla_2$; continue with $n = 3$
9	Δ_1	1	2	.	.	.	.	.	.	?*	Testing $\Delta_1, \Delta_2, \Delta_3$
10	Δ_2	.	.	.	.	5	6	.	.	✔*	* same *test* carried out in an earlier step
11	Δ_3	.	.	.	.	.	.	7	8	?*	
12	∇_1	.	.	.	.	5	6	7	8	?	Testing complements
13	∇_2	1	2	.	.	.	.	7	8	✘	$\Rightarrow$ Reduce to $c'_\times = \nabla_2$; continue with $n = 2$
14	$\Delta_1 = \nabla_2$	1	2	.	.	.	.	.	.	?*	Testing Δ_1, Δ_2
15	$\Delta_2 = \nabla_1$	.	.	.	.	.	.	7	8	?*	$\Rightarrow$ Increase granularity
16	Δ_1	1	.	.	.	.	.	.	.	?	Testing $\Delta_1, \ldots, \Delta_4$
17	Δ_2	.	2	.	.	.	.	.	.	✔	
18	Δ_3	.	.	.	.	.	.	7	.	?	
19	Δ_4	.	.	.	.	.	.	.	8	?	
20	∇_1	.	2	.	.	.	.	7	8	?	Testing complements
21	∇_2	1	.	.	.	.	.	7	8	✘	$\Rightarrow$ Reduce to $c'_\times = \nabla_2$; continue with $n = 3$
22	Δ_1	1	.	.	.	.	.	.	.	?*	Testing $\Delta_1, \ldots, \Delta_3$
23	Δ_2	.	.	.	.	.	.	7	.	?*	
24	Δ_3	.	.	.	.	.	.	.	8	?*	
25	∇_1	.	.	.	.	.	.	7	8	?	Testing complements
26	∇_2	1	.	.	.	.	.	.	8	?	
27	∇_3	1	.	.	.	.	.	7	.	?	Done
Result		1	.	.	.	.	.	7	8		

Delta Debugging: Minimality

- A test case $c \subseteq c_F$ is a *global minimum* of c_F if $\forall c' \subseteq c_F, |c'| < |c| \rightarrow test(c') \neq F$
- The global minimum is the smallest set of items that will make the program fail.
- Finding the global minimum may require us to perform an exponential number of tests.
- A test case $c \subseteq c_F$ is a *local minimum* of c_F if $\forall c' \subseteq c_F \rightarrow test(c') \neq F$
- A test-case is $c \subseteq c_F$ *n-minimal* if $\forall c' \subseteq c_F, |c| - |c'| \leq n \rightarrow test(c') \neq F$
- Delta Debugging finds an 1-minimal test case.

Delta Debugging: Monotonicity

- The super string of a failure inducing string always induces the failure.
- Delta Debugging is not effective for cases without monotonicity!

Delta Debugging: Discussion

- Scheduling decisions: Given a thread schedule for which a concurrent program works and another for which the program fails, delta debugging algorithm can narrow down the differences between two thread schedules and find the locations where a thread switch causes the program to fail.
- Chipping: Given two versions of a program such that one works correctly and the other one fails, delta debugging algorithm can be used to look for changes which are responsible for introducing the failure.
- Fault localization: apply delta debugging to memory state.
- Drawback: needs an oracle.
- Drawback: large number of runs required (potentially exponential blowup).

Table of Contents

Statistical Debugging

- Relies on a large pool of test cases (both failing and passing).
- Dynamic information from failing and passing test cases is aggregated to localize possible faulty statements.
- Output is often a list of ranked statements.

Mechanism: Tarantula

- Hypothesis: a faulty statement is more likely executed in failing runs.
- F(s)/P(s): the number of failing/passing test cases that execute statement s.
- $Suspiciousness(s) = \frac{\frac{F(s)}{|failing|}}{\frac{F(s)}{|failing|} + \frac{P(s)}{|passing|}}$

Mechanism: Tarantula (2)

- Drawback: requires a large pool of test cases (maybe not available).
- Idea: rely on deployed systems and users to provide dynamic information.
- Base dynamic information trace on predicates: branch, function returns ($== 0$, < 0, $<= 0$, ...), and scalar pairs (for each assignment $x = ...$ using variables y_i and constants c_i record $x == y_i$, $x < y_i$, ..., $x == c_i$, ...).
- Collect the evaluation of these predicates.

Statistical Analysis

- $\frac{F(p)}{S(p)}$ tells us the number of test cases in which the predicate p is true and the program fails/passes.
- $Failure(p) = \frac{F(p)}{F(p)+S(p)}$

Failure: Example

```
1 x = 10;
2 f = ...;
3 if (f==null) {
4     ... = *f;
5 }
```

- Again we have 90 passing and 10 failing test cases.
- $F(p@3) = 10$
- $S(p@3) = 0$
- $Failure(p@3) = 1$.

Context

- There are predicates that only appear in failing runs but that are themselves not faulty.
- (e.g., if the fault depends on a higher-order predicate and the code that is triggered by this predicate does not add to the fault.)
- $\frac{F(pisobserved)}{S(pisobserved)}$ how many cases in which p is executed and the program fails/passes.
- $Context(p) = \frac{F(pisobserved)}{F(pisobserved)+S(pisobserved)}$
- $Suspiciousness(p) = Failure(p) - Context(p)$

Context: Example

```
1 x = 10;
2 f = ...;
3 if (f==null) {
4   if (x>0)
5     x = 0;
6   ... = *f;
7 }
```

- Again we have 90 passing and 10 failing test cases.
- $F(p@3) = 10$, $S(p@3) = 0$, $Failure(p@3) = 1$.
- $F(p@4) = 10$, $S(p@4) = 0$, $Failure(p@4) = 1$.
- $Context(p@3) = 0.1$
- $Suspicousness(p@3) = 0.9$
- $Context(p@4) = 1$
- $Suspiciousness(p@4) = 0$

Scalability

- Distribute instrumentation across multiple versions (reduce overhead).
- Alternative: use sampling: (i) create two versions of a function (original and instrumented), (ii) use counter instead of % operation to perform sampling.

Limitations

- The faulty predicate may be evaluated to true in both failing and passing test cases (due to loops).
- (Therefore it is not found by this approach.)
- Many test cases are needed including passing/failing.
  Alternative: an oracle.
- Unclear how to handle multiple bugs or cascading bugs.
- Bug reports themselves are often not informative enough.

Questions?

?