272: Software Engineering
Fall 2018

Instructor: Tevfik Bultan

Lecture 12: Automated Debugging

Debugging

- We talked about various techniques that can be used for finding bugs in software
  - automated verification techniques
  - automated testing techniques

- As you know, after you find a bug, there is still an important activity left to do:
  - Debugging
    - today we will talk about automated debugging

Problem

- In 1999 Bugzilla, the bug database for the browser Mozilla, listed more than 370 open bugs

- Each bug in the database describes a scenario which caused software to fail
  - these scenarios are not simplified
  - they may contain a lot of irrelevant information
  - a lot of the bug reports could be equivalent

- Overwhelmed with this work, Mozilla developers sent out a call for Mozilla BugAThon call for volunteers
  - Process the bug reports by producing simplified bug reports
  - Simplifying means: turning the bug reports into minimal test cases where every part of the input would be significant in reproducing the failure

Approach

- The question is: Can one process the bug reports automatically?
- Delta debugging is an automated technique which takes a test case that causes a bug and minimizes it

An Example Bug Report

- Printing the following file causes Mozilla to crash:

```
<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
```

Continued in the next page

```
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>
```

An Example Bug Report

- It is hard to figure out what the real cause of the failure is just by staring at that file

- It would be very helpful in finding the error if we can simplify the input file and still generate the same failure

A Smaller Example

- Let's use a smaller bug report as a running example:

  When Mozilla tries to print the following HTML input it crashes:

  ```
  <SELECT NAME="priority" MULTIPLE SIZE=7>
  ```

- How can we go about simplifying this input?

  – Main Idea: Remove parts of the input and see if it still causes the program to crash

- For the above example assume that we remove characters from the input file

Bold parts remain in the input, the rest is removed

1	**<SELECT NAME="priority" MULTIPLE SIZE=7>**	F
2	<SELECT NAME="priori**ty**" **MULTIPLE SIZE=7>**	P
3	**<SELECT NAME="priori**ty" MULTIPLE SIZE=7>	P
4	<SELECT NA**ME="priority" MULTIPLE SIZE=7>**	P
5	**<SELECT NA**ME="priori**ty" MULTIPLE SIZE=7>**	F
6	**<SELECT NA**ME="priority" MULTIP**LE SIZE=7>**	F
7	**<SELECT NA**ME="priority" MULTIPLE SIZE=7>	P
8	<SELE**CT NA**ME="priority" MULTIP**LE SIZE=7>**	P
9	**<SELE**CT NAME="priority" MULTIP**LE SIZE=7>**	P
10	**<SELECT NA**ME="priority" MULTIPLE SI**ZE=7>**	F
11	**<SELECT NA**ME="priority" MULTIPLE SIZE=7>	P
12	<S**ELECT NA**ME="priority" MULTIPLE SI**ZE=7>**	P
13	**<S**EL**ECT NA**ME="priority" MULTIPLE SI**ZE=7>**	P

F means input caused failure
P means input did not cause failure (input passed)

```
14      <SELECT NAME="priority" MULTIPLE SIZE=7>      P
15      <SELECT NAME="priority" MULTIPLE SIZE=7>      P
16      <SELECT NAME="priority" MULTIPLE SIZE=7>      F
17      <SELECT NAME="priority" MULTIPLE SIZE=7>      F
18      <SELECT NAME="priority" MULTIPLE SIZE=7>      F
19      <SELECT NAME="priority" MULTIPLE SIZE=7>      P
20      <SELECT NAME="priority" MULTIPLE SIZE=7>      P
21      <SELECT NAME="priority" MULTIPLE SIZE=7>      P
22      <SELECT NAME="priority" MULTIPLE SIZE=7>      P
23      <SELECT NAME="priority" MULTIPLE SIZE=7>      P
24      <SELECT NAME="priority" MULTIPLE SIZE=7>      P
25      <SELECT NAME="priority" MULTIPLE SIZE=7>      P
26      <SELECT NAME="priority" MULTIPLE SIZE=7>      F
```

Example

- After 26 tries we found that:

  Printing an HTML file which consists of:

  `<SELECT>`

  causes Mozilla to crash.

- Delta debugging technique automates this approach of repeated trials for reducing the input.

Changes That Cause Failures

- To describe the algorithm we first need to define the process
- In general the delta debugging technique deals with *changeable circumstances*
  - circumstances whose change may cause a different program behavior
- You can think of circumstances (meaning changeable circumstances) as all the possible behaviors of the environment of the program

Circumstances and Failures

- Let E be the set of possible configurations of circumstances
  - each r $\in$ E determines a specific program run
    - If the environment is fixed the program executes deterministically

- $r_P \in$ E corresponds to a run that passes
- $r_F \in$ E corresponds to a run that fails

Changes

- We can go from one circumstance to another by changes
- A change δ is a mapping $\delta : E \rightarrow E$ which takes one circumstance and changes it to another circumstance
  - changes one program run to another program run by changing its environment (input)
- A relevant change δ is a change such that $\delta(r_P) = r_F$
  - Change δ makes the program fail

Decomposing Changes

- A change δ can be decomposed to a number of elementary changes $\delta_1, \delta_2, ..., \delta_n$ where $\delta = \delta_1 \circ \delta_2 \circ ... \circ \delta_n$
  - where $(\delta_i \circ \delta_j)(r) = \delta_i(\delta_j(r))$

- For example, deleting a part of the input file can be decomposed to deleting characters one be one from the input file
  - another way to say it: by composing deleting of single characters we can get a change that deletes part of the input file

Testing

- Given an $r \in E$ we define a function rtest: $E \rightarrow \{P, F, ?\}$
- The function rtest determines if an $r \in E$ causes the program to fail (F) or pass (P)
- The output ? means that the result is indeterminate
  - Maybe the input caused another failure other than the one we are trying to capture
- We can compute the function rtest by running the program

Test Cases

- Given a run r_P that does not cause a failure, let c denote a set of changes $\{\delta_1, \delta_2, ..., \delta_n\}$

- $c_F = \{\delta_1, \delta_2, ..., \delta_n\}$ denotes a set of changes such that
  $r_F = (\delta_1 \circ \delta_2 \circ ... \circ \delta_n)(r_P)$
- c_P is defined as $c_P = \varnothing$

- $c \subseteq c_F$ is called a test case

- To summarize
  - We have a run without failure r_P
  - We have a set of changes $c_F = \{\delta_1, \delta_2, ..., \delta_n\}$ such that
    $r_F = (\delta_1 \circ \delta_2 \circ ... \circ \delta_n)(r_P)$ where r_F is a run with failure
  - Each subset c of c_F is a test case

Testing Test Cases

- Given a test case c, we would like to know if the run generated by changing r_P by the changes in c is a run that causes a failure

- We define a function

$$test: Powerset(c_F) \rightarrow \{P, F, ?\}$$

such that, given $c=\{\delta_1, \delta_2, ..., \delta_m\} \subseteq c_F$

$$test(c) = rtest((\delta_1 \circ \delta_2 \circ ... \circ \delta_n)(r_P))$$

- Note that
  - $test(c_P) = test(\varnothing) = P$
  - $test(c_F) = test(\{\delta_1, \delta_2, ..., \delta_n\}) = F$

Minimizing Test Cases

- Now the question is: Can we find the minimal test case c such that test(c) = F?
- A test case $c \subseteq c_F$ is called the *global minimum* of c_F if
  for all $c' \subseteq c_F$, $|c'| < |c| \Rightarrow test(c') \neq F$
- Global minimum is the smallest set of changes which will make the program fail
- Finding the global minimum may require us to perform exponential number of tests

Minimizing Test Cases

- A test case $c \subseteq c_F$ is called a local minimum of c_F if
  for all $c' \subseteq c$, $test(c') \neq F$

- A test case $c \subseteq c_F$ is n-minimal if
  for all $c' \subseteq c$, $|c| - |c'| \leq n \Rightarrow test(c') \neq F$

- A test case is 1-minimal if
  for all $\delta_i \in c$, $test(c - \{\delta_i\}) \neq F$

Minimization Algorithm

- The delta debugging algorithm finds a 1-minimal test case
- It partitions the set c_F to $\Delta_1, \Delta_2, \ldots \Delta_n$
  - $\Delta_1, \Delta_2, \ldots \Delta_n$ are pairwise disjoint
  - $c_F = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$
- Define the complement of Δ_i as $\nabla_i = c_F - \Delta_i$
- Tests each test case defined by the partition and their complements
- Reduce the test case if a smaller failure inducing set is found
  - otherwise refine the partition

Each Step of the Minimization Algorithm

- Test each $\Delta_1, \Delta_2, \ldots \Delta_n$ and each $\nabla_1, \nabla_2, \ldots, \nabla_n$
- There are four possible outcomes
  1. Some Δ_i causes failure
     - Partition Δ_i to two and continue with Δ_i as the test set
  2. Some ∇_i causes failure
     - Continue with ∇_i as the test set with n – 1 subsets
  3. No test causes failure
     - Increase granularity by generating a partition with 2n subsets
  4. The granularity can no longer be increased
     - Done, found the 1-minimal subset

– In the worst case the algorithm performs $|c_F|^2 + 3|c_F|$ tests
  – For example an n character input requires n^2+3n tests in the worst case (highly unlikely to occur in practice)

Changes vs. Minimizing the Input

- Our goal is to find the smallest change δ that makes the program fail, i.e, $\delta(r_P) = r_F$

- If we pick r_P to be some trivial input (such as empty string) for which the program does not fail, then finding the smallest fault inducing change becomes equivalent to finding the smallest input that makes the program fail.

Example

- Assume that
  - Input string "`ab*defgh`" causes the program fail: r_F = `ab*defgh`
  - Program does not fail on empty string: $r_P = \varepsilon$
- If we assume that any string that contains "`*`" causes the program fail, then how will the delta-debugging algorithm behave for this case?

- In the delta-debugging formulation used by Zeller, we are trying to go from the test case which does not cause failure (i.e., empty string) to the test case that causes failure (i.e., "`ab*defgh`") with minimal changes
- Let's define δ_i to mean: "Include the i'th character of the failure inducing case in the modified input"
- Then, for our example, we have:
  - $\delta_1(\varepsilon)$ = `a`, $\delta_2(\varepsilon)$ = `b`, $\delta_3(\varepsilon)$ = `*`, $\delta_4(\varepsilon)$ = `d`, …
- Note that, the changes are composable:
  - $\delta_2(\delta_1(\varepsilon))$ = `ab`, $\delta_3(\delta_2(\delta_1(\varepsilon)))$ = `ab*`, $\delta_4(\delta_3(\delta_2(\delta_1(\varepsilon))))$ = `ab*d`, …

Example

Let's run the algorithm:

- Initially: $\Delta_1 = \{\delta_1, \delta_2, \delta_3, \delta_4\} = \nabla_2$, $\Delta_2 = \{\delta_5, \delta_6, \delta_7, \delta_8\} = \nabla_1$

test(Δ_1) = test($\{\delta_1, \delta_2, \delta_3, \delta_4\}$) = ctest($\delta_4(\delta_3(\delta_2(\delta_1(\varepsilon)))))$ = ctest(`ab*d`) = F

test(Δ_2) = test($\{\delta_5, \delta_6, \delta_7, \delta_8\}$) = ctest($\delta_5(\delta_6(\delta_7(\delta_8(\varepsilon)))))$ = ctest(`efgh`) = P

This means that we are in the case 1 of the algorithm:

1. Some Δ_i causes failure
   - Partition Δ_i to two and continue with Δ_i as the test set

Example

- We partition Δ_1 as $\{\delta_1, \delta_2\}$ and $\{\delta_3, \delta_4\}$
  - So now we have: $\Delta_1 = \{\delta_1, \delta_2\} = \nabla_2$, $\Delta_2 = \{\delta_3, \delta_4\} = \nabla_1$

$test(\Delta_1) = test(\{\delta_1, \delta_2\}) = ctest(\delta_2(\delta_1(\varepsilon))) = ctest(\texttt{ab}) = P$

$test(\Delta_2) = test(\{\delta_3, \delta_4\}) = ctest(\delta_4(\delta_3(\varepsilon))) = ctest(\texttt{*d}) = F$

This means that we are again in case 1 of the algorithm:

1. Some Δ_i causes failure
   - Partition Δ_i to two and continue with Δ_i as the test set

Example

- We partition Δ_1 as $\{\delta_3\}$ and $\{\delta_4\}$
  - So now we have: $\Delta_1 = \{\delta_3\} = \nabla_2$, $\Delta_2 = \{\delta_4\} = \nabla_1$

test(Δ_1) = test($\{\delta_3\}$) = ctest($\delta_3(\varepsilon)$) = ctest($*$) = F
test(Δ_2) = test($\{\delta_4\}$) = ctest($\delta_4(\varepsilon)$) = ctest(d) = P

We are now in case 4 of the algorithm:

4. The granularity can no longer be increased
   - Done, found the 1-minimal subset

The result is $\delta_3(\varepsilon) = *$

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 $c^{\blacksquare}_{\blacksquare} = \nabla_2$; 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 $c^{\blacksquare}_{\blacksquare} = \nabla_2$; 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 $c^{\blacksquare}_{\blacksquare} = \nabla_2$; 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		

Case Studies

- Following C program causes GCC to crash

```
#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];
}
```

Continued in the next page

```
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);
}
```

Case Studies

- The original input file 755 characters
- Delta debugging algorithm minimizes the input file to the following file with 77 characters

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

- If a single character is removed from this file then it does not induce the failure

Case Studies

- Reducing a Mozilla bug to
  - Print the html file containing `<SELECT>`
- Minimizing fuzz input
  - Fuzzing: Take a program, send it randomly generated input and see it crashes
  - Used delta debugging to minimize fuzz input

Isolating Failure Inducing Differences

- Instead of minimizing the input that causes the failure we can also try to isolate the differences that cause the failure
  - Minimization means to make each part of the simplified test case relevant: removing any part makes the failure go away
  - Isolation means to find one relevant part of the test case: removing this particular part makes the failure go away
- For example changing the input from

  `<SELECT NAME="priority" MULTIPLE SIZE=7>`

  to

  `SELECT NAME="priority" MULTIPLE SIZE=7>`

  makes the failure go away
  - This means that inserting the character < is a failure inducing difference
- Delta debugging algorithm can be modified to look for minimal failure inducing differences

Failure Inducing Differences: Example

- Changing the input program for GCC from the one on the left to the one on the right removes the failure

This input causes failure

```
#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];
}
```

This input does not cause failure

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

Modified statement is shown in box

Delta Debugging for Program Changes

- Finding failure inducing code changes in programs
  - 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

- Main challenge is dealing with inconsistency: We do not want the output to be indeterminate (?)
  - For example, applying a subset of the changes can cause the program to fail at compilation resulting in ? as the outcome
  - To deal with this, it might be necessary to change the granularity of the changes or grouping related changes

Delta Debugging for Program Changes

- GNU DDD (DataDisplayDebugger) 3.1.2 dumps core when invoked with the name of a non-existing file while its predecessor DDD 3.1.1 simply gives an error message
- There were 116 changes between 3.1.1 and 3.1.2
- These changes were split in to 344 textual changes to the DDD source
- Running the delta debugging algorithm finds the failure-inducing change in after 31 test runs

Delta Debugging for Program Changes

- To reduce inconsistencies:
  - Changes were grouped according to the date they were applied
  - It was assumed that each change implied all earlier changes
- Then using delta debugging algorithm in 12 test runs the following failure inducing difference was reported:

```
diff …
< string classpath =
<       getenv("CLASSPATH") != 0 ? getenv("CLASSPATH") : "."
---
> string classpath = source_view->classpath();
```

- When called with an argument that is not a file name DDD 3.1.1 checks whether it is a Java class, so DDD consults its environment for the class lookup path
- As an "improvement" DDD 3.1.2 uses a dedicated method for this purpose, but `source-view` pointer is initialized only later, causing a core dump.

Delta Debugging for Cause-Effect Chains

- Delta debugging can be used to minimize the difference in fault-inducing input
- But even a small change in the input can cause a large change in the program behavior
- Can we use delta debugging to identify the cause of the fault in the program behavior?
- Basic idea: Apply delta debugging on program states in order to isolate the variables and values that are relevant for the failure
  - These isolated variables constitute the cause-effect chain that leads from the root cause to the failure

Delta Debugging for Cause-Effect Chains

- Try to minimize the differences between the successful run and the failing run in terms of the program states (which consist of variable-value pairs)
- This has to be done carefully since changing the program state can cause program to crash
- Uses a data structure called memory graph that represents all values and variables but also represents operations like variable access, pointer dereferencing, struct member access, or array element access by edges
- Memory graphs represent the entire state of the program and thus avoid problems due to incomplete comparison of program states

Delta Debugging for Thread Schedules

- Isolating failure inducing thread schedules
  - 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

A Selection of Papers on Delta Debugging

- “Simplifying and Isolating Failure-Inducing Input”. Andreas Zeller and Ralf Hildebrandt; IEEE Transactions on Software Engineering 28(2), February 2002, pp. 183-200.
- “Yesterday, my program worked. Today, it does not. Why?” Andreas Zeller; Proc. ESEC/FSE 99, September 1999, Vol. 1687 of LNCS, pp. 253-267
- "Isolating Cause-effect Chains from Computer Programs" Andreas Zeller. 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2002).
- “Isolating Failure-Inducing Thread Schedules.” J.-D. Choi, A. Zeller. Proc. International Symposium on Software Testing and Analysis (ISSTA 2002), July 2002