DEBUGGING: DELTA DEBUGGING

WS 2018/2019

Martina Seidl
Institute for Formal Models and Verification
Simplifying the Problem

- problem found
  ⇒ simplify it
  - which circumstances are relevant?
  - which circumstances can be omitted?
- turn problem report into concise test case
  (relevant details only)
- by adding and removing circumstances
  (experimentally)

**delta debugging**: automated debugging method for systematically simplifying test cases such that the problem still occurs
How and Why To Simplify

How?

- by experimentation, one finds out whether a circumstance is relevant or not:
  1. omit the circumstance and try to reproduce the problem
  2. the circumstance is relevant if the problem no longer occurs

Why?

- easier communication
- easier debugging
- easier identification of duplicates
Basic Idea of Delta Debugging

from https://www.st.cs.uni-saarland.de/dd/
Delta Debugging Roadmap

1. identify the test case(s)
2. identify the deltas
3. set up a Delta Debugging framework
   □ implement a reduction strategy (binary search)
4. write a testing function
   □ test automatically if failure occurs under simplified test case
5. invoke Delta Debugging
Delta Debugging: General Approach

**binary search:**

1. remove half of the input
2. check if the output is still wrong
   - 2.1 yes: further simplify
   - 2.2 no: reset the state and remove other half of the input
A Delta Debugging Algorithm: Preliminaries

- **elements:**
  - circumstance: $\delta$
  - all circumstances: $C = \{\delta_1, \delta_2, \ldots\}$
  - configuration: $c \subseteq C$, (e.g., $c = \{\delta_1, \ldots, \delta_n\}$)

- **tests**
  - testing function: $\text{test}(c) \in \{\checkmark, \times, \, ?\}$
  - failure inducing configuration: $\text{test}(c_x) = \times$
  - relevant configuration: $c'_x \subseteq c_x$ such that
    $$\forall \delta_i \in c'_x : \text{test}(c'_x \setminus \{\delta_i\}) \neq \times$$
A Delta Debugging Algorithm: Binary Strategy

- split input: $c'_x = c_1 \cup c_2$
- if removing $c_1$ results in failure:
  \[ \text{test}(c'_x \setminus c_1) = \times \Rightarrow c'_x = c_x \setminus c_1 \]
- if removing $c_2$ results in failure:
  \[ \text{test}(c'_x \setminus c_2) = \times \Rightarrow c'_x = c_x \setminus c_2 \]
- otherwise: increase granularity
  \[ c'_x = c_1 \cup c_2 \cup c_3 \cup c_4 \]

general strategy: split test case into $n$ parts (initially 2)
The ddmin Algorithm

- result: $c'_x = \text{ddmin}(c_x)$
  - $c'_x$ is a relevant configuration
  - $c'_x \subseteq c_x$

- implementation: $\text{ddmin}(c'_x) = \text{ddmin}'(c'_x, 2)$

\[
\text{ddmin}'(c'_x, n) =
\]

- if $|c'_x| = 1$
  - return $c'_x$
- if $(\text{test}(c'_x \setminus c_i) = x)$ for some $i \in \{1..n\}$
  - return $\text{ddmin}'(c'_x \setminus c_i, \max(n - 1, 2))$
- if $n < |c'_x|$
  - return $\text{ddmin}'(c'_x, \min(2n, |c'_x|))$
- otherwise
  - $c'_x$
Optimizations

- caching
- stop when no progress is observed
  - after a certain time
  - after a certain number of unsuccessful simplifications
  - when a certain granularity has been reached
- syntactic simplifications
- isolation of differences instead of circumstances
Example: ddSMT

1 (set-logic UFNIA)
2 (declare-sort sort1 0)
3 (declare-fun x () sort1)
4 (declare-fun y () sort1)
5 (assert (= x y ))
6 (push 1)
7 (define-sort sort2 () Bool)
8 (declare-fun x () sort2)
9 (declare-fun y () sort2)
10 (assert (and (as x Bool) (as y Bool)))
11 (assert (! (not (as x Bool)) :named z))
12 (assert z)
13 (pop 1)
14 (assert (forall ((z Int)) (exists ((zz Int)) (= z zz))))
15 (check-sat)
16 (get-value ((let ((x 1) (y 1)) (= x y))))
17 (exit)
Example: ddSMT

```bash
#!/bin/sh

if [ \"grep -c \"<get-value\>\" \$1\" -ne 0 \];
  then exit 1
fi

exit 0
```

→ simulates: SMT Solver does not support `get-value` commands
Example: ddSMT

1 (set-logic UFNIA)
2 (declare-sort sort1 0)
3 (declare-fun x () sort1)
4 (declare-fun y () sort1)
5 (assert (= x y ))

6 (push 1)
7   (define-sort sort2 () Bool)
8   (declare-fun x () sort2)
9   (declare-fun y () sort2)
10  (assert (and (as x Bool) (as y Bool)))
11  (assert (! (not (as x Bool)) :named z))
12  (assert z)
13 (pop 1)
14  (assert (forall ((z Int)) (exists ((zz Int)) (= z zz))))
15 (check-sat)
16  (get-value ((let ((x 1) (y 1)) (= x y))))
17 (exit)

example by Aina Niemetz [SMT13]
Example: ddSMT

1 (set-logic UFNIA)
2 (declare-sort sort1 0)
3 (declare-fun x () sort1)
4 (declare-fun y () sort1)
5 (assert (= x y ))
6 (push 1)
7 (define-sort sort2 () Bool)
8 (declare-fun x () sort2)
9 (declare-fun y () sort2)
10 (assert (and (as x Bool) (as y Bool)))
11 (assert (! (not (as x Bool)) :named z))
12 (assert z)
13 (pop 1)
14 (assert (forall ((z Int)) (exists ((zz Int)) (= z zz))))
15 (check-sat)
16 (get-value ((let ((x 1) (y 1)) (= 0 0))))
17 (exit)

all variable bindings substituted

example by Aina Niemetz [SMT13]
Example: ddSMT

1 (set-logic UFNIA)
2 (declare-sort sort1 0)
3 (declare-fun x () sort1)
4 (declare-fun y () sort1)

15 (check-sat)
16 (get-value ((= 0 0)))
17 (exit)
Example: ddSMT

1 (set-logic UFNIA)
2 (declare-sort sort1 0)
3 (declare-fun x () sort1)
4 (declare-fun y () sort1)
5 (assert (= x y ))
6 (push 1)
7 (define-sort sort2 () Bool)
8 (declare-fun x () sort2)
9 (declare-fun y () sort2)
10 (assert (and (as x Bool) (as y Bool)))
11 (assert (! (not (as x Bool)) :named z))
12 (assert z)
13 (pop 1)
14 (assert (forall ((z Int)) (exists ((zz Int)) (= z zz))))
15 (check-sat)
16 (get-value ((= 0 0)))
17 (exit)

non-constant Boolean term

15 (check-sat)
16 (get-value ((= 0 0)))
17 (exit)
Example: ddSMT

1 (set-logic UFNIA)
2 (declare-sort sort1 0)
3 (declare-fun x () sort1)
4 (declare-fun y () sort1)

5 (assert (= x y ))
6 (push 1)
7 (define-sort sort2 () Bool)
8 (declare-fun x () sort2)
9 (declare-fun y () sort2)
10 (assert (and (as x Bool) (as y Bool)))
11 (assert (! (not (as x Bool)) :named z))
12 (assert z)
13 (pop 1)
14 (assert (forall ((z Int)) (exists ((zz Int)) (= z zz))))
15 (check-sat)
16 (get-value (false))
17 (exit)
Example: ddSMT

1 (set-logic UFNIA)

2 (declare-sort sort1 0)
3 (declare-fun x () sort1)
4 (declare-fun y () sort1)

5 (assert (= x y ))
6 (push 1)
7 (define-sort sort2 () Bool)
8 (declare-fun x () sort2)
9 (declare-fun y () sort2)
10 (assert (and (as x Bool) (as y Bool)))
11 (assert (! (not (as x Bool)) :named z))
12 (assert z)
13 (pop 1)
14 (assert (forall ((z Int)) (exists ((zz Int)) (= z zz))))
15 (check-sat)
16 (get-value (false))
17 (exit)

example by Aina Niemetz [SMT13]
Example: ddSMT

1 (set-logic UFNIA)

16 (get-value (false))
17 (exit)