DEBUGGING: DELTA DEBUGGING

WS 2017/2018

Martina Seidl
Institute for Formal Models and Verification

JOHANNES KEPLER
UNIVERSITY LINZ

Simplifying the Problem

- problem found

  $\Rightarrow$ 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

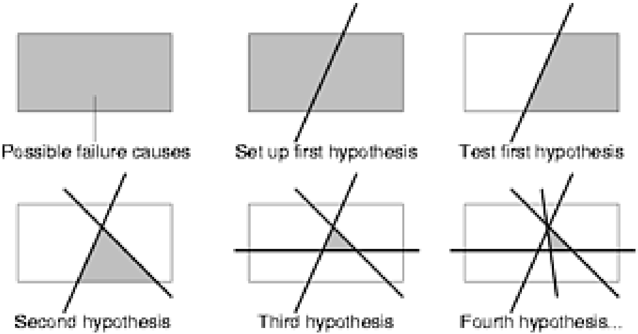
How?

- 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: δ
  - all circumstances: $C = \{\delta_1, \delta_2, ...\}$
  - configuration: $c \subseteq C$, (e.g., $c = \{\delta_1, ..., \delta_n\}$)
- tests
  - testing function: test(c) $\in \{$✓, ✗, ?$\}$
  - failure inducing configuration: test($c_{✗}$) = ✗
  - relevant configuration: $c'_{✗} \subseteq c_{✗}$ such that
    $\forall \delta_i \in c'_{✗}$: test($c'_{✗} \setminus \{\delta_i\}$) $\neq$ ✗

JKU

A Delta Debugging Algorithm: Binary Strategy

- split input: $c'_{✗} = c_1 \cup c_2$
- if removing c_1 results in failure:

$$\text{test}(c'_{✗} \setminus c_1) = ✗ \Rightarrow c'_{✗} = c_{✗} \setminus c_1$$

- if removing c_2 results in failure:

$$\text{test}(c'_{✗} \setminus c_2) = ✗ \Rightarrow c'_{✗} = c_{✗} \setminus c_2$$

- otherwise: increase granularity

$$c'_{✗} = c_1 \cup c_2 \cup c_3 \cup c_4$$

general strategy: split test case into n parts (initially 2)

The ddmin Algorithm

- result: $c'_{✗} = \text{ddmin}(c_{✗})$
  - $c'_{✗}$ is a relevant configuration
  - $c'_{✗} \subseteq c_{✗}$
- implementation: $\text{ddmin}(c'_{✗}) = \text{ddmin}'(c'_{✗}, 2)$

$\text{ddmin}'(c'_{✗}, n) =$

if $\lvert c'_{✗} \rvert = 1$	return $c'_{✗}$
if (test($c'_{✗} \setminus c_i$) = ✗ for some $i \in \{1..n\}$)	return $\text{ddmin}'(c'_{✗} \setminus c_i, max(n-1, 2))$
if $n < \lvert c'_{✗} \rvert$	return $\text{ddmin}'(c'_{✗}, min(2n, \lvert c'_{✗} \rvert))$
otherwise	$c'_{✗}$

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

```
(set-logic UFNIA)
(declare-sort sort1 0)
(declare-fun x () sort1)
(declare-fun y () sort1)
(assert (= x y ))
(push 1)
  (define-sort sort2 () Bool)
  (declare-fun x () sort2)
  (declare-fun y () sort2)
  (assert (and (as x Bool) (as y Bool)))
  (assert (!  (not (as x Bool)) :named z))
  (assert z)
(pop 1)
(assert (forall ((z Int)) (exists ((zz Int)) (= z zz))))
(check-sat)
(get-value ((let ((x 1) (y 1)) (= x y))))
(exit)
```

example by Aina Niemetz [SMT13]

Example: ddSMT

```sh
#!/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 by Aina Niemetz [SMT13]

Example: ddSMT

```
(set-logic UFNIA)
(declare-sort sort1 0)
(declare-fun x () sort1)
(declare-fun y () sort1)
(assert (= x y ))
(push 1)
  (define-sort sort2 () Bool)
  (declare-fun x () sort2)
  (declare-fun y () sort2)
  (assert (and (as x Bool) (as y Bool)))
  (assert (!  (not (as x Bool)) :named z))
  (assert z)
(pop 1)
(assert (forall ((z Int)) (exists ((zz Int)) (= z zz))))
(check-sat)
(get-value ((let ((x 1) (y 1)) (= x y))))
(exit)
```

redundant

Example: ddSMT

```
(set-logic UFNIA)
(declare-sort sort1 0)
(declare-fun x () sort1)
(declare-fun y () sort1)
```

all variable bindings substituted

```
(check-sat)
(get-value ((let ((x 1) (y 1)) (= 0 0))))
(exit)
```

Example: ddSMT

```
(set-logic UFNIA)
(declare-sort sort1 0)
(declare-fun x () sort1)
(declare-fun y () sort1)

(check-sat)
(get-value ((= 0 0)))
(exit)
```

Example: ddSMT

```
(set-logic UFNIA)
(declare-sort sort1 0)
(declare-fun x () sort1)
(declare-fun y () sort1)
```

non-constant Boolean term

```
(check-sat)
(get-value ((= 0 0)))
(exit)
```

example by Aina Niemetz [SMT13]

Example: ddSMT

```
(set-logic UFNIA)
(declare-sort sort1 0)
(declare-fun x () sort1)
(declare-fun y () sort1)

(check-sat)
(get-value (false))
(exit)
```

Example: ddSMT

```
(set-logic UFNIA)
(declare-sort sort1 0)
(declare-fun x () sort1)
(declare-fun y () sort1)
```

redundant

```
(check-sat)
(get-value (false))
(exit)
```

Example: ddSMT

```
(set-logic UFNIA)

(get-value (false))
(exit)
```

example by Aina Niemetz [SMT13]