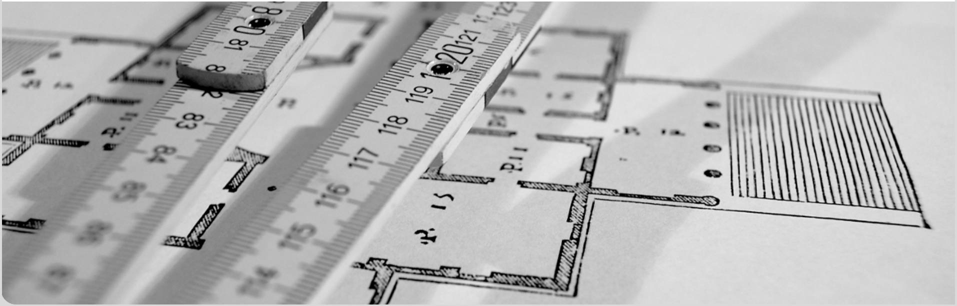
Karlsruher Institut für Technologie

Delta Debugging

A summary of Delta Debugging and its uses

Moritz Laupichler | 19. November 2018

FAKULTÄT FÜR INFORMATIK



Motivation

«Everyone knows that debugging is twice as hard as writing a program in the first place.

So if you're as clever as you can be when you write it, how will you ever debug it?»

– Brian Kernighan in "The Elements of Programming Style"

Version Control allows DD

- Version Control has been around since the 80's
- central terms: configuration, change

The DD idea

Configuration:	Changes:	Configuration:
Yesterday	$\longrightarrow$	Today
	$\longrightarrow$	
Passes tests. ✓	$\longrightarrow$	Tests fail. ✗

Idea: Delta Debugging

Find the minimal set of changes between Yesterday and Today that induces the failure.

The intuitive approach

- $\mathcal{C} = \{\Delta_1, \ldots, \Delta_n\}$: All changes between Yesterday and Today
- $c \subseteq \mathcal{C}$: A configuration (set of changes applied to Yesterday)
- $test : 2^{\mathcal{C}} \rightarrow \{✓, ✗, ?\}$: Result of the tests applied to a configuration

A simple binary search can be conducted to find simple failure inducing changes:

function SIMPLEDD($c : 2^{\mathcal{C}}$)
  if $\mid c \mid = 1$ **then return** c
  Split c into two halves c_1, c_2 so that $c_1 \cap c_2 = \emptyset$
  if ($test(c_1) = ✗$) **then return** $simpledd(c_1)$
  else return $simpledd(c_2)$

Difficulty #1: Interference

- *ddsimple* works for single failure inducing-changes.
  - In each recursion step it applies only the set of changes known to contain a failure inducing change.
- But what if two changes exist that individually pass the tests but their combination induces failure?

Difficulty #1: Interference

Let $c_1, c_2 \in \mathcal{C}$. c_1 and c_2 **interfere** when $\mathit{test}(c_1) = ✓$, $\mathit{test}(c_2) = ✓$ but $\mathit{test}(c_1 \cup c_2) = ✗$.

Difficulty #1: Interference

Idea: Leave one set of changes applied

If a configuration $c = c_1 \cup c_2$ with $test(c_1) = ✓$ and $test(c_2) = ✓$ is found by *simpledd* interference between c_1 and c_2 has been detected. Run *simpledd* on c_1 while leaving c_2 applied and vice versa.

function $dd_2(c, r : 2^{\mathcal{C}}) : 2^{\mathcal{C}}$

let $c_1, c_2 \subseteq c$ with $c_1 \cup c_2 = c, c_1 \cap c_2 = \emptyset, |c_1| \approx |c_2|$

return $\begin{cases} c & \text{if } |c| = 1, \\ dd_2(c_1, r) & \text{if } test(c_1 \cup r) = ✗, \\ dd_2(c_2, r) & \text{if } test(c_2 \cup r) = ✗, \\ dd_2(c_1, c_2 \cup r) \cup dd_2(c_2, c_1 \cup r) & \text{otherwise} \end{cases}$

Difficulty #2: Inconsistency

- *dd* combines changes arbitrarily (in the case of interference)
- That can lead to inconsistent configurations, i.e. no test outcome can be determined for these configurations

Difficulty #2: Inconsistency

Let $c_1, c_2 \subseteq \mathcal{C}$. An **inconsistency** occurs when $test(c_1 \cup c_2) = ?$.

Idea: More granular subsets

If less changes are applied at once the chances of an inconsistent result are reduced. Hence, if the algorithm cannot find any consistent confgurations reduce the number of changes per subset.

Difficulty #2: Inconsistency

Necessary changes to *dd*:

1. Extend *dd* to work on a number n of subsets $c_1, \ldots, c_n$
2. **Interference** occurs when c_i and its complement $\bar{c}_i$ both pass: $test(c_i) = ✓$ and $test(\bar{c}_i) = ✓$ ($\bar{c}_i = \mathcal{C} \setminus c_i$)
3. Add the case of **preference**: If $test(c_i) = ?$ and $test(\bar{c}_i) = ✓$ we deduce that c_i contains a failure inducing change.
4. Add the case of **Try again**: In any other case repeat the process with $2n$ subsets.

The dd+ algorithm

Solves it all!

Some case study

Shows how awesome DD is.

DD as an abstract method

Foundation or part of different solutions.