Testing, Debugging, Program Verification

Debugging Programs, Part I

Wolfgang Ahrendt, Vladimir Klebanov, Moa Johansson

7 November 2011

Testing and Debugging

- Classical view of testing (first block of course):
  - Uncover yet unknown problems.
  - Cover as many potential problems as possible.
    Recall: Coverage criteria from last lecture.
- Debugging:
  - Known problems.
  - Test-case to reproduce the problem.
  - Tests rerun many times to simplify and understand problem.
  - Regression testing: Ensure fix worked and did not break anything else!

Debugging Block

Today:

- Bug reports may involve large inputs to large systems.
- Controlling large programs: Which component is buggy?
- Can we find a minimal test-case displaying the bug?

Next lecture:

- Chasing bugs: Find the source using Debuggers and Logging.

Motivation

Debugging is unavoidable and a major economical factor

- Software bugs cost the US economy ca. 60 billion US$/y (2002)
  In general estimated 0.6% of the GDP of industrial countries
- Ca. 80 percent of software development costs spent on identifying and correcting defects
- Software re-use is increasing and tends to introduce bugs due to changed specification in new context (Ariane 5)

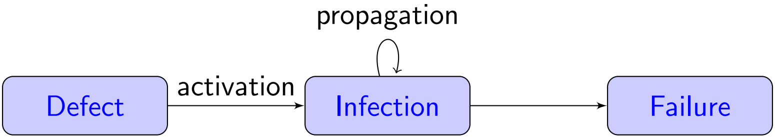
Debugging needs to be systematic

- Bug reports may involve large inputs
- Programs may have thousands of memory locations
- Programs may pass through millions of states before failure occurs

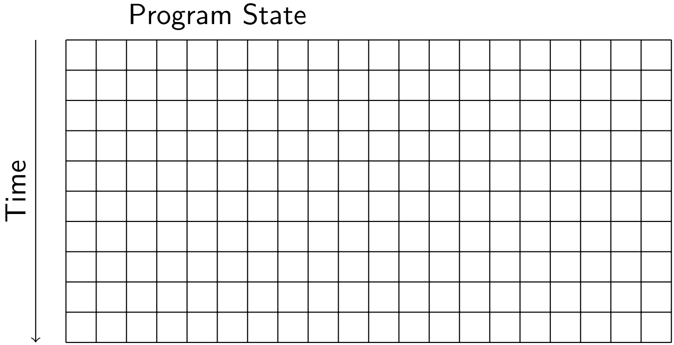
Reminder: Terminology

Bug-Related Terminology

1. Defect (aka bug, fault) introduced to the code by programmer
   Not always programmer's fault: changing/unforeseen requirements
2. Defect may cause infection of the program state during execution
   Not all defects cause an infection: e.g., Pentium bug
3. An infected state propagates during execution
   Infected parts of states may be overwritten, corrected, unused
4. An infection may cause a failure: an externally observable error
   This may include non-termination

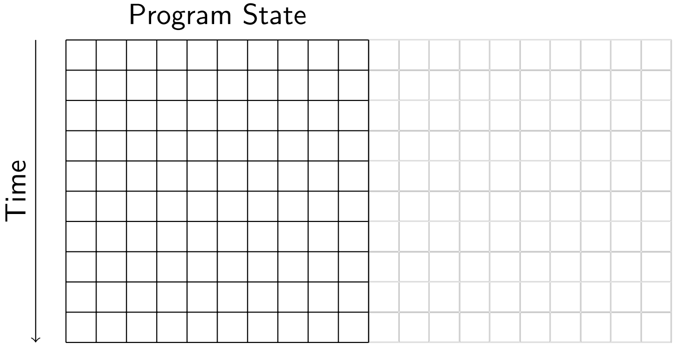


The Main Steps in Systematic Debugging



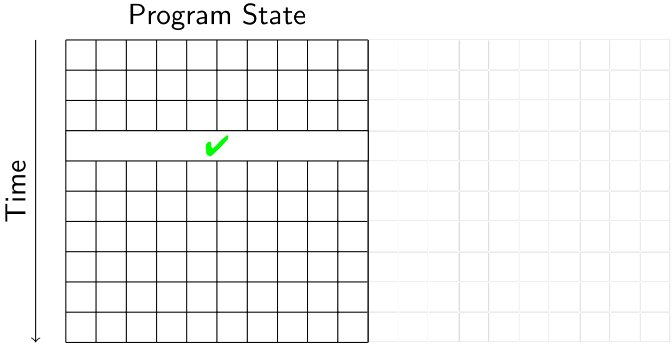
Reproduce failure with test input

The Main Steps in Systematic Debugging



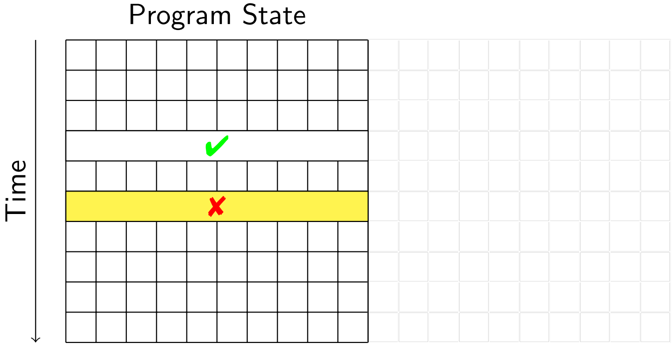
Reduction of failure-inducing problem

The Main Steps in Systematic Debugging



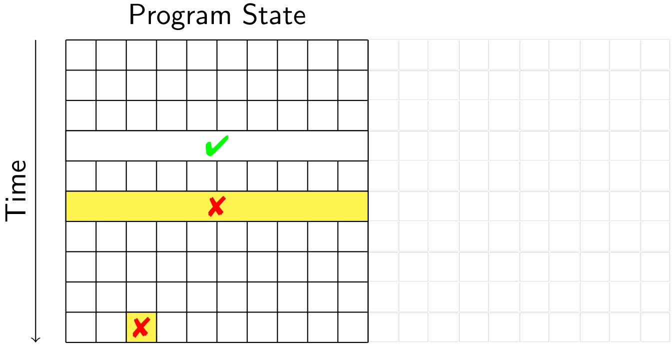
State known to be sane

The Main Steps in Systematic Debugging



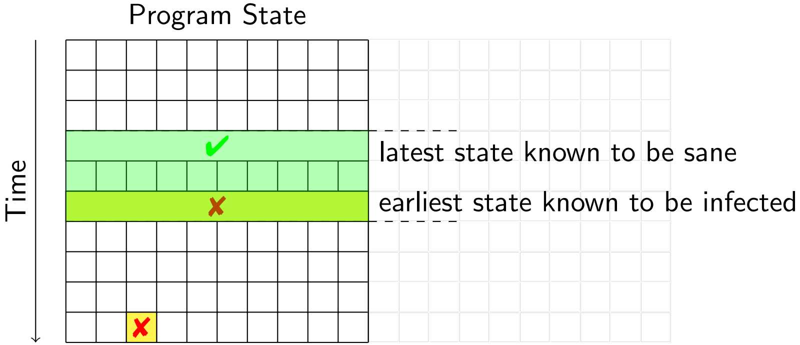
State known to be infected

The Main Steps in Systematic Debugging



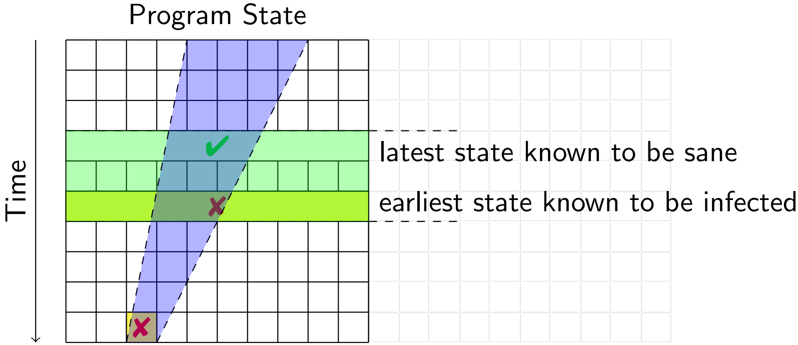
State where failure becomes observable

The Main Steps in Systematic Debugging



- Separate sane from infected states

The Main Steps in Systematic Debugging



- Separate sane from infected states
- Separate relevant from irrelevant states

Debugging Techniques

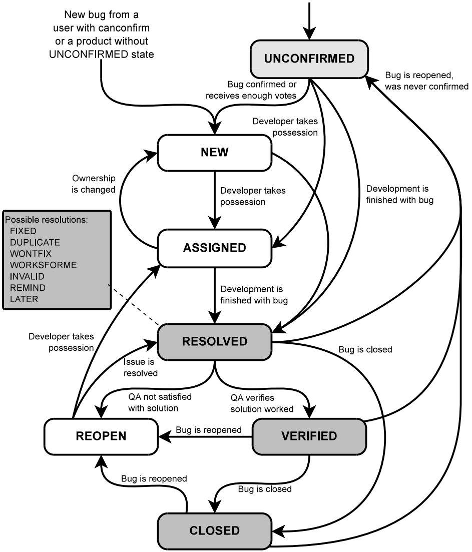
The analysis suggests main techniques used in systematic debugging:

- Bug tracking — Which start states cause failure?
- Program control — Design for Debugging
- Input simplification — Reduce state size
- State inspection using debuggers
- Tracking causes and effects — From failure to defect

Common Themes

- Fighting combinatorial explosion: separate relevant from irrelevant
- Being systematic: avoid repetition, ensure progress, use tools

Bug Tracking: Bug Life Cycle



http://www.bugzilla.org/docs/

From Bug to Test Case

Scenario

Assume MOZILLA crashes while printing a certain URL to file

We need to turn the bug report into an automated test case!

Why?

- Write a test to reproduce the problem
- Write a test to simplify the problem
- Run a test to observe the run
- Run a test to validate a fix
- Re-run tests to protect against regression

From Bug to Test Case

Scenario

Assume MOZILLA crashes while printing a certain URL to file

We need to turn the bug report into an automated test case!

Automated test case execution essential

- Reproduce the bug reliably (cf. scientific experiment)
- Repeated execution necessary during isolation of defect
- After successful fix, bug must become part of nightly-run test suite

Prerequisites for automated execution

1. Program control (without manual interaction)
2. Isolating small program units that contain the bug

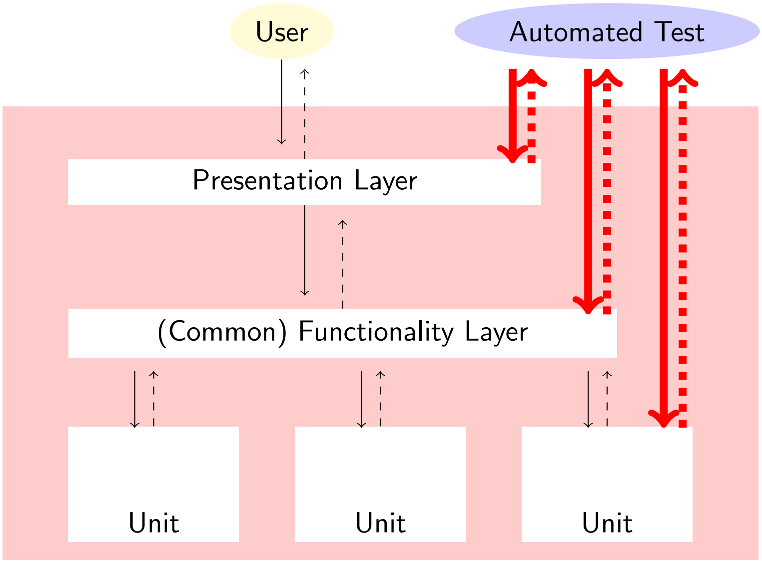
Program Control

Enable automated run of program that may involve user interaction

Example (Sequence of interaction that led to the crash)

1. Launch MOZILLA
2. Open URL location dialogue
3. Type in a location
4. Open Print dialogue
5. Enter printer settings
6. Initiate printing

Program Interfaces for Testing



Automated Testing at Different Layers

Presentation Scripting languages for capturing & replaying user I/O

- Specific to an OS/window system/hardware
- Scripts tend to be brittle (depend e.g. on screen layout)

Functionality Interface scripting languages

1. Implementation-specific scripting languages: VBScript
2. Universal scripting languages with application-specific extension: Python, Perl, Tcl

Unit Unit testing frameworks (as in previous lecture)

- JUnit, CPPUnit, VBUnit, ...

Testing Layers: Discussion

The higher the layer, the more difficult becomes automated testing

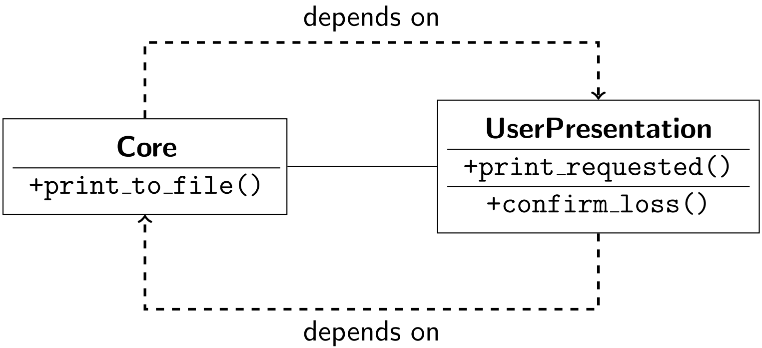
- Scripting may vary with OS/window system/prog. lang.
- Test scripts depend on (for example):
  - application environment (printer driver)
  - hardware (screen size), work environment (paper size)

Test at the unit layer whenever possible!

Requires modular design with low coupling

- Good design is essential even for testing and debugging!
- We concentrate on decoupling rather than specific scripts

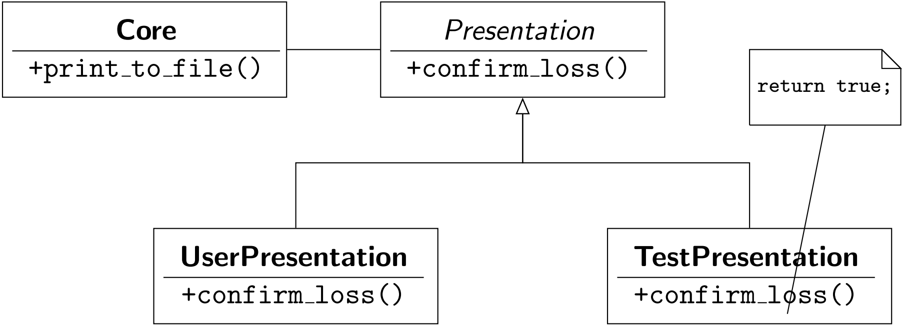
Disentangling Layers



Circular Layer Dependency

- Presentation depends on core functionality
  calls `print_to_file()` to do the printing
- Functionality depends on presentation
  calls `confirm_loss()` to prevent accidental file removal

Breaking Circular Dependencies by Refactoring



- Programming to interfaces important even for testability
- More general: Model-View-Controller Pattern

Isolating Units

Use test interfaces to isolate smallest unit containing the defect

- In the Mozilla example, unit for file printing easily identified
- In general, use debugger (see next lecture) to trace execution

Mini Quiz: Designing for Debugging

- Why is it difficult to debug a program through the Presentation Layer (i.e. user interface, GUI)?
  Need scripts (can be brittle), often OS dependent, hardware dependent etc...
- What does it mean for a program to have *low coupling*?
  Less dependecy between modules, e.g. less shared data.
- Why is this beneficial for debugging?
  Easier to test individual units in isolation.
- How should you design your program to achieve this?
  - Use interfaces, implement several presentation layers, e.g. user-presentation and debug-presentation.
  - Model-view-controller design pattern.

From Bug to Test Case, Part II

Scenario

Assume MOZILLA crashes while printing a loaded URL to file

We need to turn the bug report into an automated test case!

We managed to isolate the relevant program unit, but ...

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD␣HTML␣4.01//EN">
<html lang="en">

<head>
 <title>Mozilla.org</title>
 <meta http-equiv="Content-Type"
       content="text/html;␣charset=UTF-8">
... ca 200 lines more
```

Problem Simplification

We need a small failed test case

Divide-and-Conquer

1. Cut away one half of the test input
2. Check, whether one of the halves still exhibits failure
3. Continue until minimal failing input is obtained

Problems

- Tedious: rerun tests manually
- Boring: cut-and-paste, rerun
- What, if none of the halves exhibits a failure?

Automatic Input Simplification

- Automate cut-and-paste and re-running tests
- Partition test input into n chunks

c_1	$\cdots$	c_n

- Remove one chunk at a time, re-run test on remaining pattern

c_1	$\cdots$	c_{i-1}	c_{i+1}	$\cdots$	c_n

- Increase granularity (number of chunks) when no failure occurs

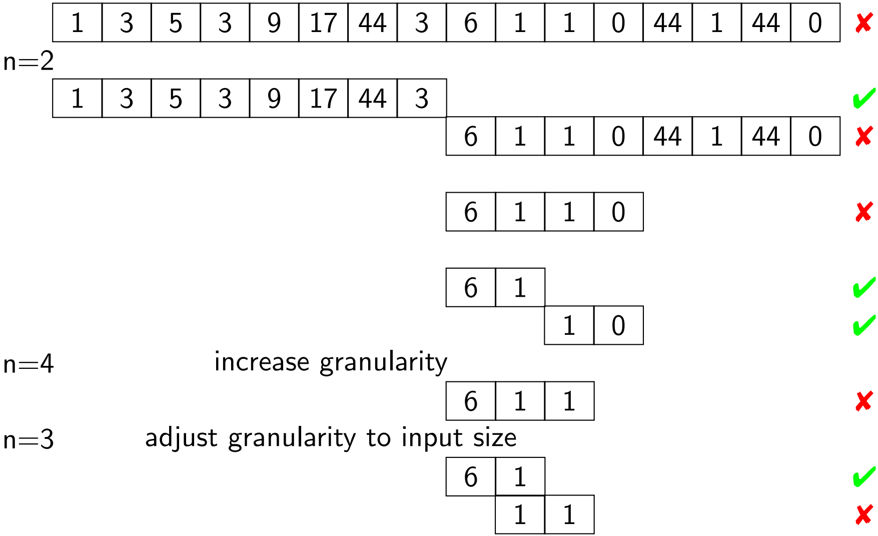
Example

public static int `checkSum(`**int**`[] a)`

- is supposed to compute the checksum of an integer array
- gives wrong result, whenever `a` contains two identical consecutive numbers, but we don't know that yet
- we have a failed test case, e.g., from protocol transmission:

$\{$`1,3,5,3,9,17,44,3,6,1,1,0,44,1,44,0`$\}$

Input Simplification ($n =$ number of chunks)



Simplification Algorithm

Prerequisites

- Let $c_{\text{✘}}$ be an input configuration (sequence of individual inputs)
- test(c) runs a test on c with possible outcome ✔, ✘, ?

Find 1-minimal failing input: call $\text{ddMin}(c_0, 2)$ with $\text{test}(c_0) = $ ✘

$$\text{ddMin}(c_{\text{✘}}, n) = \begin{cases} c_{\text{✘}} & |c_{\text{✘}}| = 1 \\ \text{ddMin}(c_{\text{✘}} \setminus c, \max(n-1, 2)) & \text{test}(c_{\text{✘}} \setminus c_i) = \text{✘ for some } c_i \in c_{\text{✘}} \\ \text{ddMin}(c_{\text{✘}}, \min(2n, |c_{\text{✘}}|)) & \text{otherwise, if } n < |c_{\text{✘}}| \\ c_{\text{✘}} & \text{otherwise} \end{cases}$$

At each step, split $c_{\text{✘}}$ as $c_1, \ldots, c_n$

Minimal Failure Configuration

- Minimization algorithm is easy to implement
- Realizes input size minimization for failed run
- Implementation:
  - Small program in your favorite PL (Zeller: PYTHON, JAVA)
  - Eclipse plugin DDINPUT at `http://www.st.cs.uni-sb.de/eclipse/`
- Demo: `DD.java`, `Dubbel.java`

Consequences of Minimization

- Input small enough for observing, tracking, locating (next topics)
- Minimal input often provides important hint for source of defect

Principal Limitations of Input Minimization

Algorithm does not find all failing configurations with minimal size

Computes failure-inducing input configuration that is 1-minimal:
Taking away any individual input removes the failure

1. Finds only first failing 1-minimal config (no backtracking)
   There could be other, shorter 1-minimal configs
2. Misses failing configs created by taking away several chunks

Example (Incompleteness of minimization)

Failure occurs for integer array when frequency of occurrences of all identical numbers is even:

{1,2,1,2} fails
Taking away any chunk of size 1 or 2 passes
{1,1} fails, too, and is even smaller

Inefficiency of Linear Minimization

Minimization algorithm ignores any structure of input

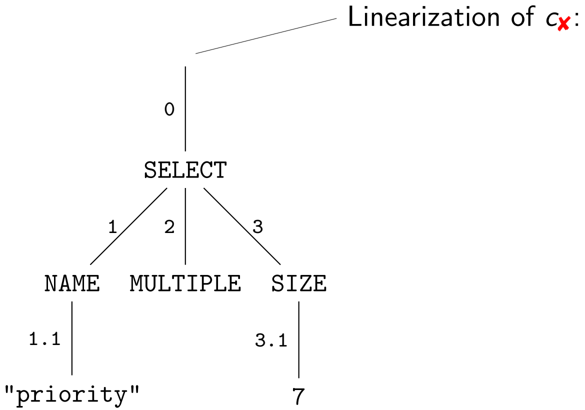
Example (.html input configuration)

<**SELECT NAME**="priority"**MULTIPLE SIZE**=7> ✘

- Most substrings are not valid HTML: test result ? ("unresolved")
- There is no point to test beneath granularity of tokens

Minimization may require a unnecessarily large number of steps

Structured Minimization

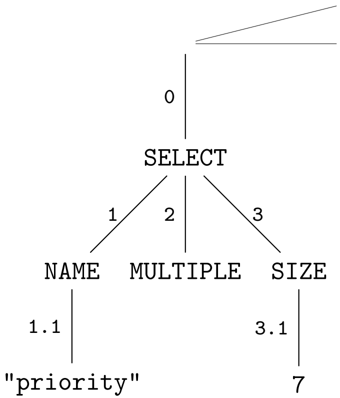


Input configuration: nodes in abstract syntax tree, not characters

Structured Minimization

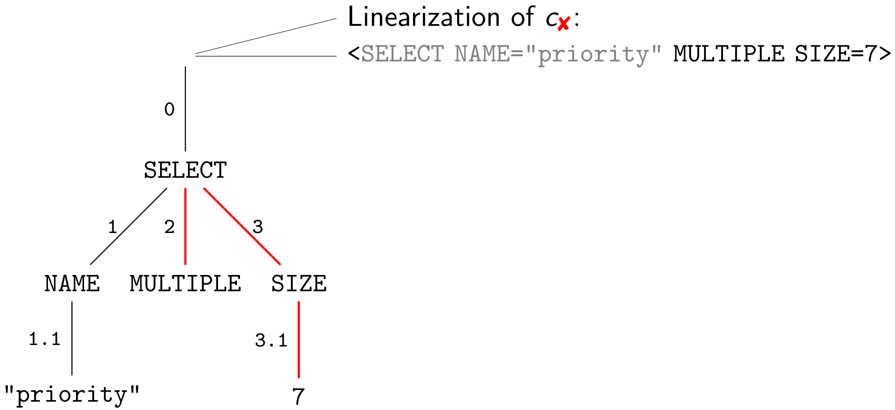
Linearization of $c_{✘}$:

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



$c_{✘} = \{0, 1, 1.1, 2, 3, 3.1\}$

Structured Minimization

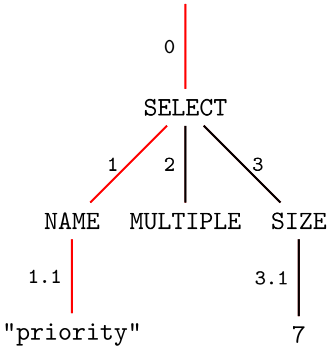


$c_{✗} = \{0, 1, 1.1, 2, 3, 3.1\}$ infeasible (not a tree), return ?

Structured Minimization

Linearization of $c_{\text{✘}}$:

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

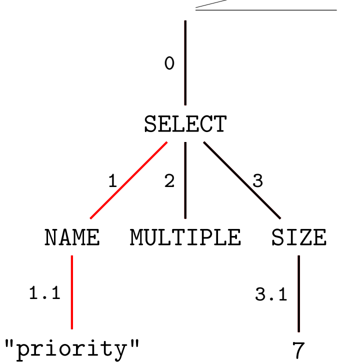


$c_{\text{✘}} = \{0, 1, 1.1, 2, 3, 3.1\}$ Failure occurs, reduce length

Structured Minimization

Linearization of $c_{✘}$:

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

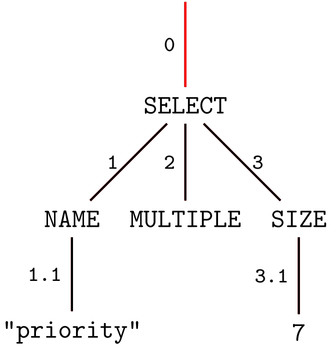


$c_{✘} = \{0, 1, 1.1, 2, 3, 3.1\}$ infeasible (not well-formed HMTL), return ?

Structured Minimization

Linearization of $c_{✘}$:

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



$c_{✘} = \{0, 1, 1.1, 2, 3, 3.1\}$ Failure occurs, can't be minimized further

Delta Debugging, Adaptive Testing

The Bigger Picture

- Minimization of failure-inducing input is instance of delta debugging
- Delta debugging is instance of adaptive testing

Delta Debugging

Isolating failure causes by narrowing down differences ("Δ") between runs.

This principle used in various debugging activities

Adaptive Testing

A test series where each test depends on the outcome of earlier tests

Some Tips

Logging

Log all debugging activities, particularly, test cases and outcomes

Add Testing Interfaces

Avoids interaction (tedious!) and presentation-layer scripts (brittle!)

Fix Time Limit for Quick-and-Dirty Debugging

Use "naive" debugging when bug seems obvious, but 10 mins max!

Test the Right Program

Is the path and filename correct? Did you compile?

What Next?

- ✔ Bug tracking
- ✔ Program control — Design for Debugging
- ✔ Input simplification

- Execution observation
  - With logging
  - Using debuggers
- Tracking causes and effects

Literature for this Lecture

Essential

Zeller Why Programs Fail: A Guide to Systematic Debugging
2nd edition, Morgan Kaufmann, 2009
Chapters 2, 3, 5

Available online as e-books via Chalmers library, navigate to "E-book collections", "Books24x7", and register

Background

McConnell Code Complete: A Practical Handbook for Software Construction, 2nd edition, Microsoft Press, 2004
Chapter 23