Software Engineering

Lecture 11: Testing and Debugging — Debugging

Peter Thiemann

University of Freiburg, Germany

10.06.2013

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 specifications in new context (Ariane 5)

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 specifications 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

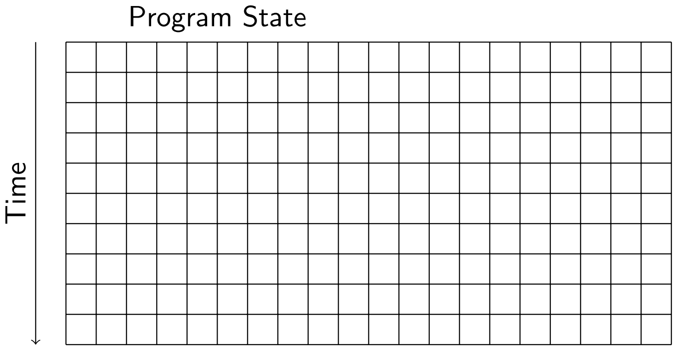
Terminology

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. Infection may cause a failure: externally observable error
   May include non-termination

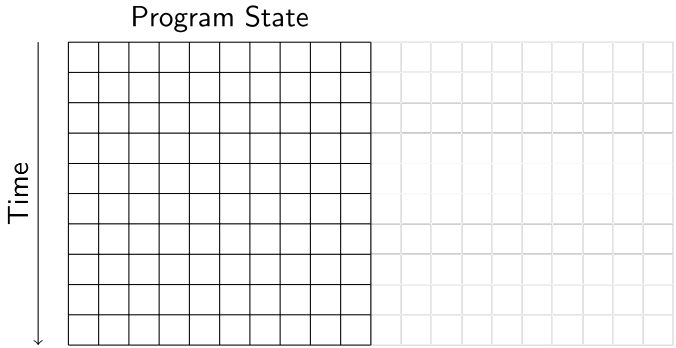
Defect — Infection — Propagation — Failure

The Main Steps in Systematic Debugging



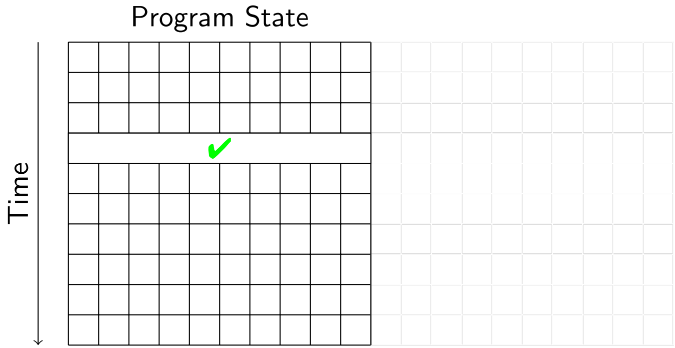
Reproduce failure with test input

The Main Steps in Systematic Debugging



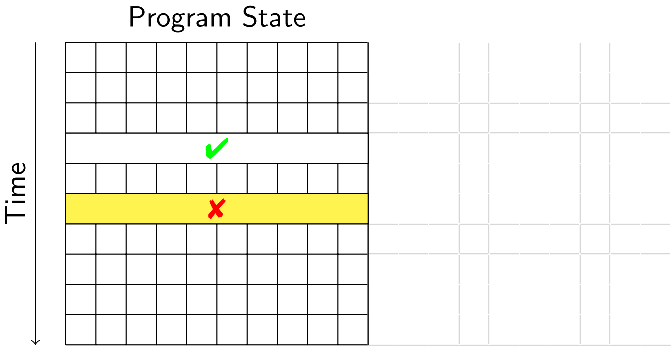
Reduction of failure-inducing problem

The Main Steps in Systematic Debugging



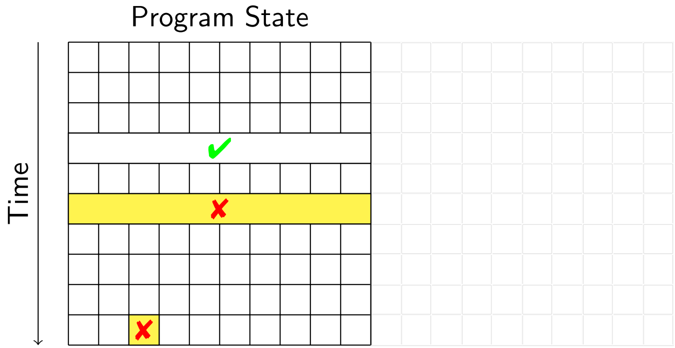
State known to be healthy

The Main Steps in Systematic Debugging



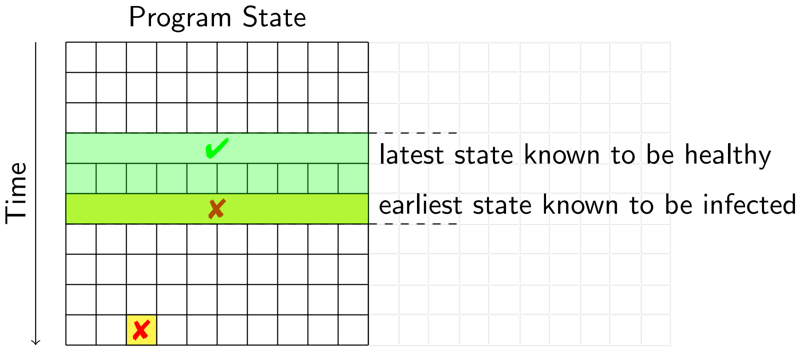
State known to be infected

The Main Steps in Systematic Debugging



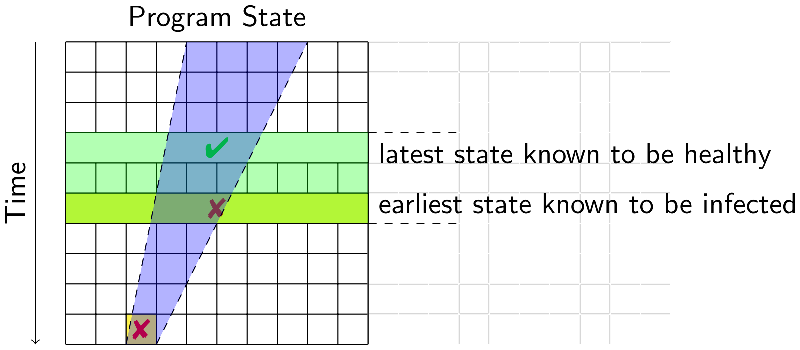
State where failure becomes observable

The Main Steps in Systematic Debugging



- Separate healthy from infected states

The Main Steps in Systematic Debugging



- Separate healthy from infected states
- Separate relevant parts from irrelevant ones

Debugging Techniques

The analysis suggests main techniques used in systematic debugging:

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

Debugging Techniques

The analysis suggests main techniques used in systematic debugging:

- Bug tracking — Which initial states cause failure?
- Program control — Design for Debugging
- Input simplification — Reduce state space
- State observation and watching 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 Tracking Life Cycle

Unconfirmed

Raw problem report, often from end user

Bug Tracking Life Cycle

Unconfirmed ➜ New

Description complete, no duplicate of existing bug

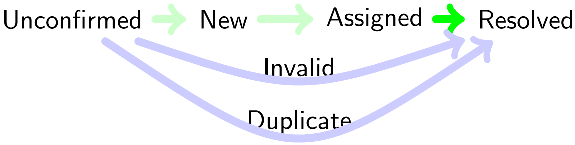
Bug Tracking Life Cycle

Bug Tracking Life Cycle

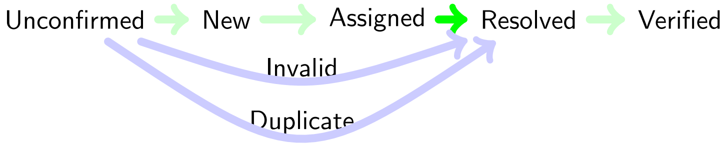
Unconfirmed → New → Assigned → Resolved

Tag	Meaning
Fixed Problem	is fixed
WontFix	Impossible or undesirable ("feature") to fix
WorksForMe	Can't be reproduced
Invalid	Not a problem or insufficiently described
Duplicate	Refers to existing problem

Bug Tracking Life Cycle

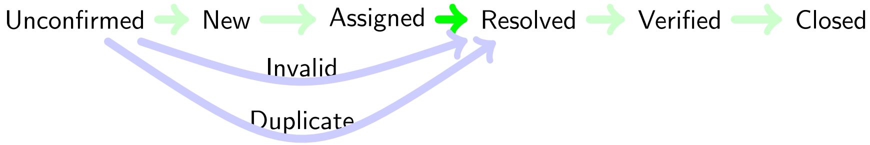


Bug Tracking Life Cycle



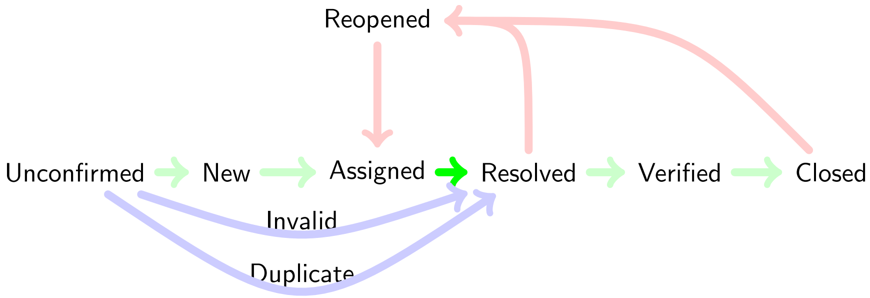
Only if Fixed

Bug Tracking Life Cycle



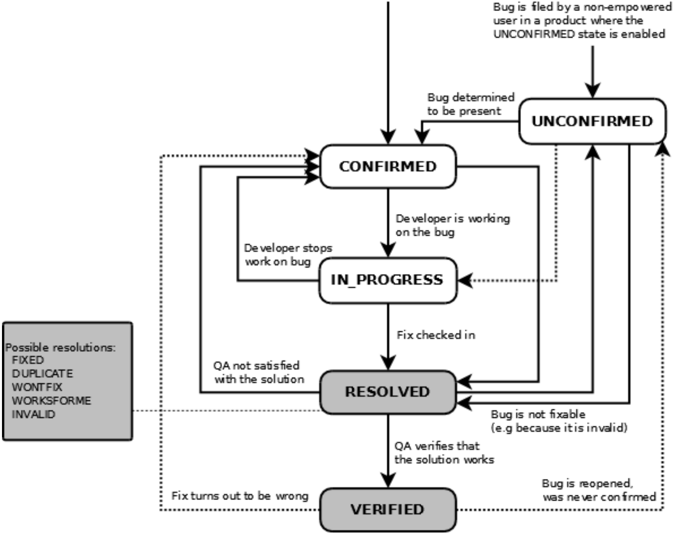
Important to avoid cluttering of bug database

Bug Tracking Life Cycle



The fix didn't work after all . . .

Bugzilla's Bug Lifecycle



From Bug to Test Case

Program Control: From Bug to Test Case

Scenario

Assume Firefox crashes while printing a certain URL to file

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

Program Control: From Bug to Test Case

Scenario

Assume FIREFOX 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, test case becomes part of test suite

Prerequisites for automated execution

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

Program Control

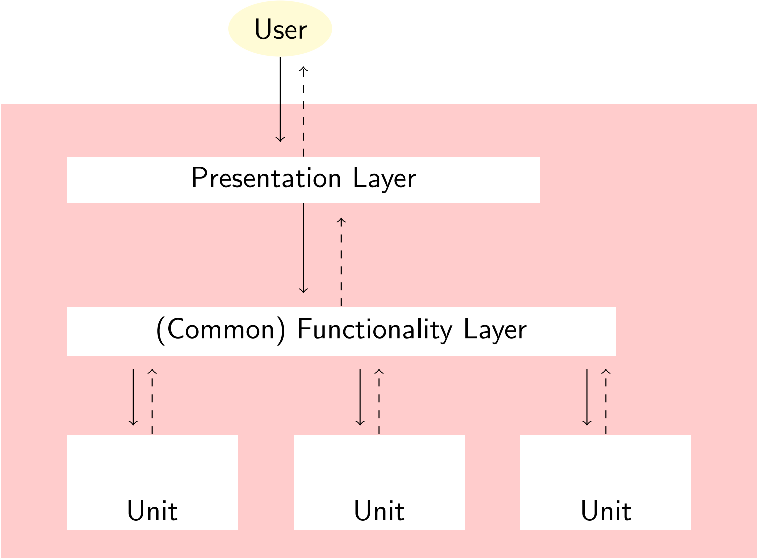
Program Control

Enable automated run of program that may involve user interaction

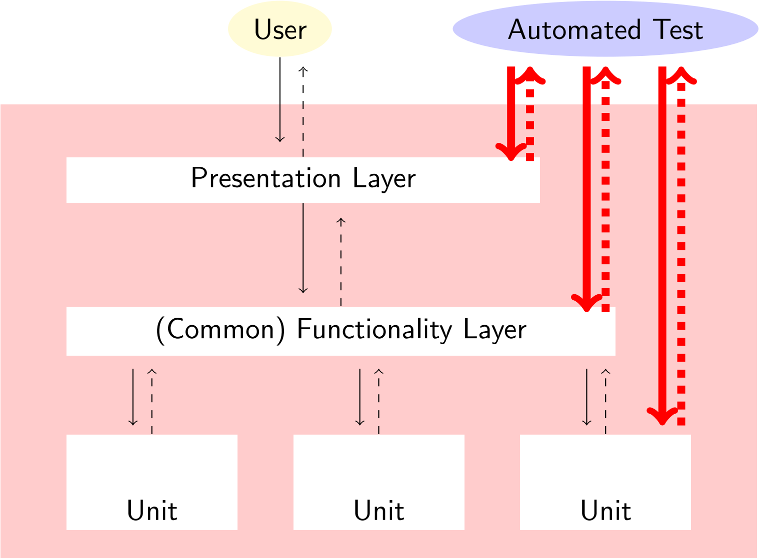
Example (Sequence of interaction that led to the crash)

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

Alternative Program Interfaces for Testing



Alternative 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

Functionality

Interface scripting languages

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

Unit testing frameworks (as in previous lecture)

JUnit, CPPUnit, VBUnit, . . .

Testing Layers: Discussion

The higher the layer, the more difficult becomes automated testing

- Scripting languages specific to OS/Window S./Progr. L.
- Test scripts depend on (for example):
  - application environment (printer driver)
  - hardware (screen size), working environment (paper size)

Testing Layers: Discussion

The higher the layer, the more difficult becomes automated testing

- Scripting languages specific to OS/Window S./Progr. L.
- Test scripts depend on (for example):
  - application environment (printer driver)
  - hardware (screen size), working environment (paper size)

Test at the unit layer whenever possible!

Testing Layers: Discussion

The higher the layer, the more difficult becomes automated testing

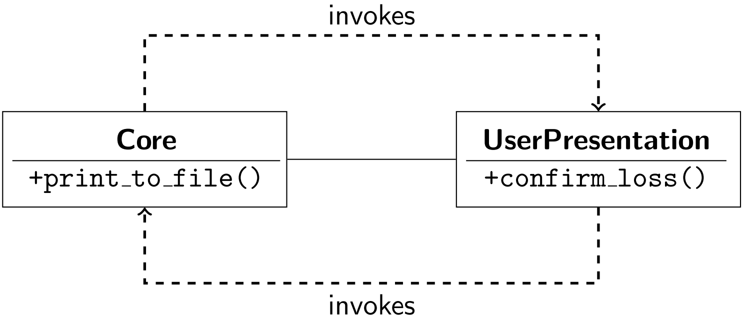
- Scripting languages specific to OS/Window S./Progr. L.
- Test scripts depend on (for example):
  - application environment (printer driver)
  - hardware (screen size), working environment (paper size)

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

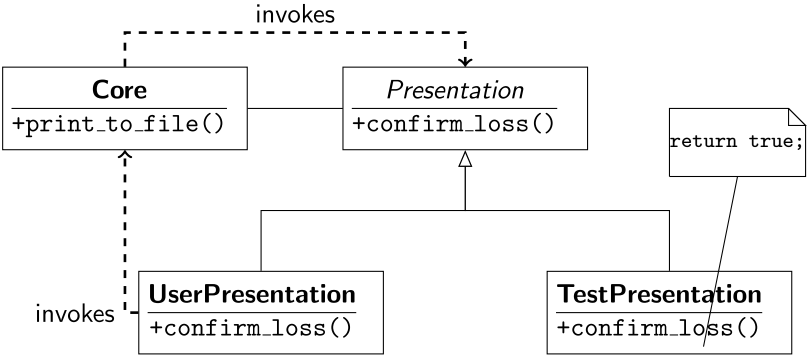
Disentangling Layers



Circular Dependency Example

- Print-to-file is core functionality
  calls `confirm_loss()` to prevent accidental file removal
- Override-if-file-exists question is in UI
  relies on core functionality to check file existence

Breaking Circular Dependencies by Refactoring



- Programming to interfaces important even for testability

Isolating Units

Use test interfaces to isolate smallest unit containing the defect

- In the Firefox example, unit for file printing easily identified
- In general, use debugger to trace execution

Problem Simplification

From Bug to Test Case, Part II

Scenario

Assume FIREFOX 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 . . .

From Bug to Test Case, Part II

Scenario

Assume FIREFOX 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">
... about 200 lines more
```

Problem Simplification

We need a small test case that fails!

Problem Simplification

We need a small test case that fails!

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

Problem Simplification

We need a small test case that fails!

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
- Increase granularity of chunks when no failure occurs

Automatic Input Simplification

- Automate cut-and-paste and re-running tests
- Increase granularity 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, eg, from transmission trace:

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

Input Simplification ($n =$ number of chunks)

1	3	5	3	9	17	44	3	6	1	1	0	44	1	44	0

✘

Input Simplification ($n =$ number of chunks)

1	3	5	3	9	17	44	3	6	1	1	0	44	1	44	0	✘

n=2

1	3	5	3	9	17	44	3	✔

Input Simplification ($n =$ number of chunks)

1	3	5	3	9	17	44	3	6	1	1	0	44	1	44	0	✘

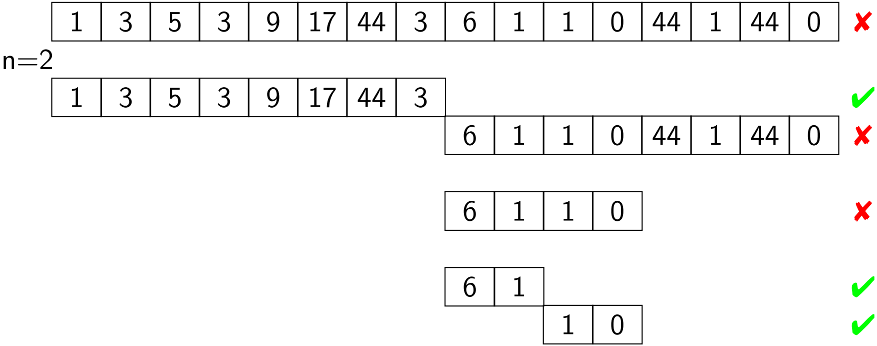
n=2

1	3	5	3	9	17	44	3									✔
								6	1	1	0	44	1	44	0	✘

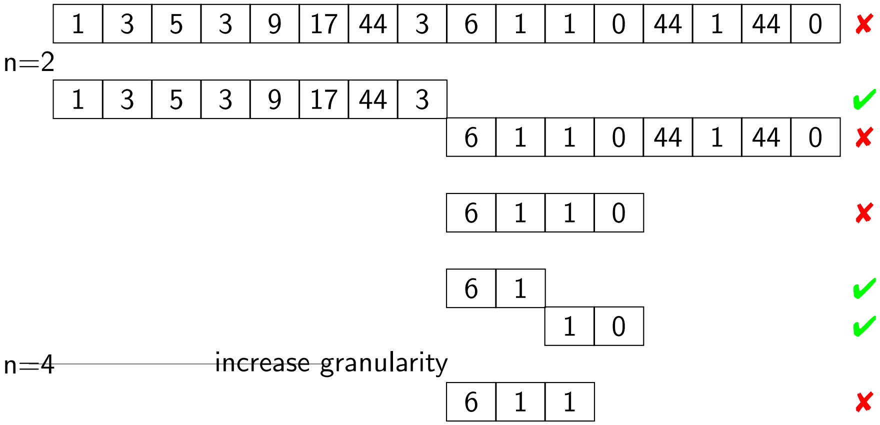
Input Simplification ($n =$ number of chunks)

	1	3	5	3	9	17	44	3	6	1	1	0	44	1	44	0	✘
n=2																	
	1	3	5	3	9	17	44	3									✔
									6	1	1	0	44	1	44	0	✘
									6	1	1	0					✘

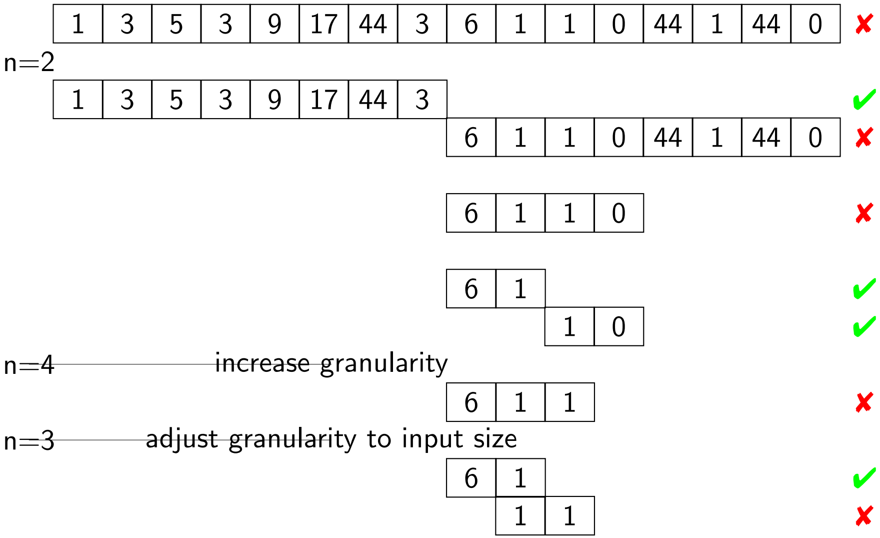
Input Simplification ($n =$ number of chunks)



Input Simplification (n = number of chunks)



Input Simplification ($n =$ number of chunks)



Simplification Algorithm — Delta Debugging

Prerequisites

- $\text{test}(c) \in \{✔, ✘, ?\}$ runs a test on configuration c
- Let $c_{✘}$ be a failing input configuration with
  - $\text{test}(c_{✘}) = ✘$
  - length $l = |c_{✘}|$ if $c_{✘} = \{x_1, \ldots, x_l\}$
  - view at granularity $n \leq l$: $c_{✘} = c_1 \cup \cdots \cup c_n$, $c_i \neq \emptyset$
  - write $c_i \in_n c$

Simplification Algorithm — Delta Debugging

Prerequisites

- $\text{test}(c) \in \{✔, ✘, ?\}$ runs a test on configuration c
- Let $c_{✘}$ be a failing input configuration with
  - $\text{test}(c_{✘}) = ✘$
  - length $l = |c_{✘}|$ if $c_{✘} = \{x_1, \ldots, x_l\}$
  - view at granularity $n \leq l$: $c_{✘} = c_1 \cup \cdots \cup c_n$, $c_i \neq \emptyset$
  - write $c_i \in_n c$

Find minimal failing input: call $\text{dd}_{Min}(c_0, 2)$ with $\text{test}(c_0) = ✘$

$$\text{dd}_{Min}(c_{✘}, n) = \begin{cases} c_{✘} & |c_{✘}| = 1 \\ \text{dd}_{Min}(c_{✘} - c, \max(n-1, 2)) & c \in_n c_{✘} \wedge \text{test}(c_{✘} - c) = ✘ \\ \text{dd}_{Min}(c_{✘}, \min(2n, |c_{✘}|)) & n < |c_{✘}| \\ c_{✘} & \text{otherwise} \end{cases}$$

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 `www.st.cs.uni-sb.de/eclipse/`
-

Demo: `DD.java`, `Dubbel.java`

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 `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 computes minimal failure-inducting subsequence of the input:

  Taking away any chunk of any length removes the failure
- However, there may be failing inputs with smaller size!
  1. Algorithm investigates only one failing input of smaller size
  2. Misses failure-inducing inputs created by taking away several chunks

Principal Limitations of Input Minimization

- Algorithm computes minimal failure-inducting subsequence of the input:

  Taking away any chunk of any length removes the failure
- However, there may be failing inputs with smaller size!
  1. Algorithm investigates only one failing input of smaller size
  2. Misses failure-inducing inputs created by taking away several chunks

Example (Incompleteness of minimization)

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

Principal Limitations of Input Minimization

- Algorithm computes minimal failure-inducting subsequence of the input:

  Taking away any chunk of any length removes the failure
- However, there may be failing inputs with smaller size!
  1. Algorithm investigates only one failing input of smaller size
  2. Misses failure-inducing inputs created by taking away several chunks

Example (Incompleteness of minimization)

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

{1,2,1,2} fails

Principal Limitations of Input Minimization

- Algorithm computes minimal failure-inducting subsequence of the input:

  Taking away any chunk of any length removes the failure
- However, there may be failing inputs with smaller size!
  1. Algorithm investigates only one failing input of smaller size
  2. Misses failure-inducing inputs created by taking away several chunks

Example (Incompleteness of minimization)

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

{1,2,1,2} fails
Taking away any chunk of size 1 or 2 passes

Principal Limitations of Input Minimization

- Algorithm computes minimal failure-inducting subsequence of the input:

  Taking away any chunk of any length removes the failure
- However, there may be failing inputs with smaller size!
  1. Algorithm investigates only one failing input of smaller size
  2. Misses failure-inducing inputs created by taking away several chunks

Example (Incompleteness of minimization)

Failure occurs for integer array when frequency of occurrences of all 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

Limitations of Linear Minimization

Minimization algorithm ignores structure of input

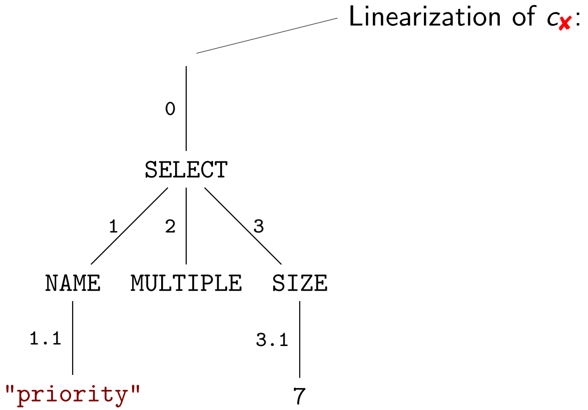
Example (.html input configuration)

```
<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 very large number of steps

Structured Minimization

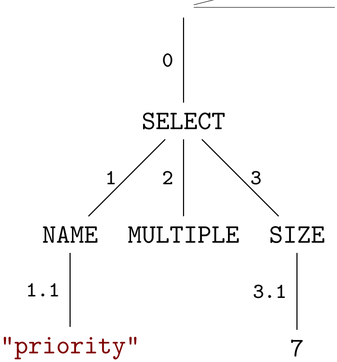


Input configuration consists of nodes in ABS not characters

Structured Minimization

Linearization of $c_{✗}$:

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

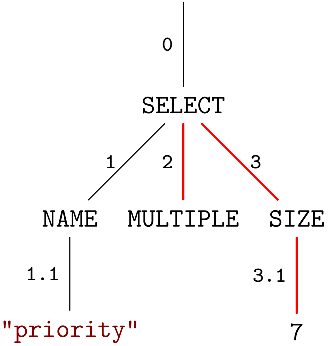


$c_{✗} = \{0,\ 1,\ 1.1,\ 2,\ 3,\ 3.1\}$

Structured Minimization

Linearization of $c_{\text{✘}}$:

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

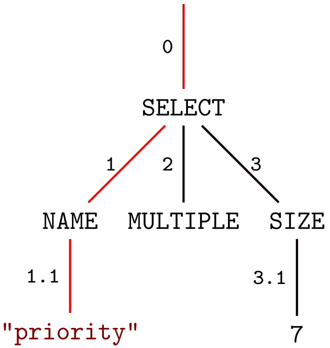


$c_{\text{✘}} = \{0, 1, 1.1, 2, 3, 3.1\}$ infeasible (not a tree) return ?

Structured Minimization

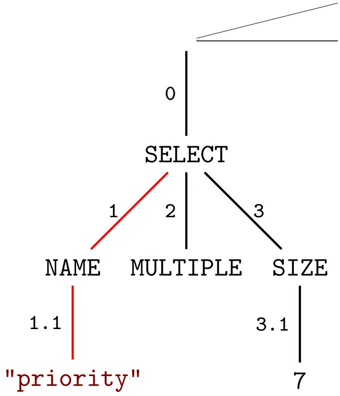
Linearization of $c_{\text{✘}}$:

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



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

Structured Minimization



Linearization of $c_{✘}$:

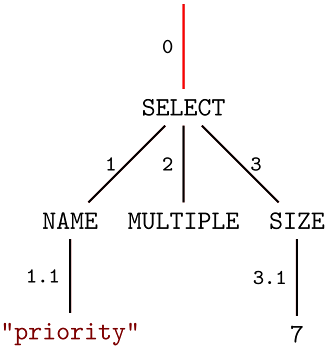
<SELECT NAME="priority" MULTIPLE SIZE=7>

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

Structured Minimization

Linearization of $c_{\text{✘}}$:

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



$c_{\text{✘}} = \{0, 1, 1.1, 2, 3, 3.1\}$ 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, Adaptive Testing

The Bigger Picture

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

Definition (Delta Debugging)

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

This principle is used in various debugging activities

Delta Debugging, Adaptive Testing

The Bigger Picture

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

Definition (Delta Debugging)

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

This principle is used in various debugging activities

Definition (Adaptive Testing)

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

What Next?

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

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, Morgan Kaufmann, 2005
Chapters 2, 3, 5

Background

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