DEBUGGING

WS 2017/2018

Martina Seidl
Institute for Formal Models and Verification

JOHANNES KEPLER
UNIVERSITY LINZ

The Program “Middle”

```
int middle (int x, int y, int z) {
  int m = z;
  if (y < z) {
    if (x < y)
      m = y;
    else if (x < z)
       m = y;
  } else {
    if (x > y)
      m = y;
    else if (x > z)
      m = x;
  }
  return m;
}
```

This program is supposed to return the middle (median) of three numbers.

JKU

The Program “Middle”

```
int middle (int x, int y, int z) {
  int m = z;
  if (y < z) {
    if (x < y)
      m = y;
    else if (x < z)
       m = y;
  } else {
    if (x > y)
      m = y;
    else if (x > z)
      m = x;
  }
  return m;
}
```

Some test cases:

```
middle (1, 2, 3) = 2
middle (1, 3, 2) = 2
middle (2, 3, 1) = 2
middle (3, 1, 2) = 2
middle (3, 2, 1) = 2
middle (1, 1, 1) = 1
middle (1, 1, 2) = 1
middle (1, 2, 1) = 1
middle (2, 1, 1) = 1
middle (1, 2, 2) = 2
middle (2, 1, 2) = 2
middle (2, 2, 1) = 2
middle (2, 1, 3) = 1
```

The Program “Middle”

```
int middle (int x, int y, int z) {
  int m = z;
  if (y < z) {
    if (x < y)
      m = y;
    else if (x < z)
       m = y;
  } else {
    if (x > y)
      m = y;
    else if (x > z)
      m = x;
  }
  return m;
}
```

Some test cases:

```
middle (1, 2, 3) = 2
middle (1, 3, 2) = 2
middle (2, 3, 1) = 2
middle (3, 1, 2) = 2
middle (3, 2, 1) = 2
middle (1, 1, 1) = 1
middle (1, 1, 2) = 1
middle (1, 2, 1) = 1
middle (2, 1, 1) = 1
middle (1, 2, 2) = 2
middle (2, 1, 2) = 2
middle (2, 2, 1) = 2
middle (2, 1, 3) = 1
```

BUG!

The Program “Middle”

```
int middle (int x, int y, int z) {
  int m = z;
  if (y < z) {
    if (x < y)
      m = y;
    else if (x < z)
→      m = y;
  } else {
    if (x > y)
      m = y;
    else if (x > z)
      m = x;
  }
  return m;
}
```

Some test cases:

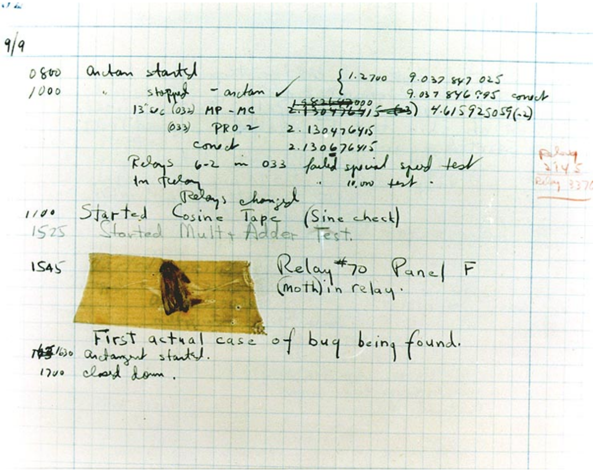
```
middle (1, 2, 3) = 2
middle (1, 3, 2) = 2
middle (2, 3, 1) = 2
middle (3, 1, 2) = 2
middle (3, 2, 1) = 2
middle (1, 1, 1) = 1
middle (1, 1, 2) = 1
middle (1, 2, 1) = 1
middle (2, 1, 1) = 1
middle (1, 2, 2) = 2
middle (2, 1, 2) = 2
middle (2, 2, 1) = 2
middle (2, 1, 3) = 1
```

BUG!

JKU

The First Documented Bug in a Computer

moth in Harvard Mark II found on Sept 9, 1947



Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988

From Defects to Failures

1. **The programmer creates a defect.**

   Why? Who is to blame?

2. **The defect causes an infection.**

   After the execution of the defect,
   the program state might not be as intended.

3. **The infection propagates.**

   Or it can be overwritten, masked, or corrected
   by later program instructions.

4. **The infection causes a failure**, i.e., an observable error in the program behavior.

From Defects to Failures

1. **The programmer creates a defect.**
   Why? Who is to blame?
2. **The defect causes an infection.**
   After the execution of the defect,
   the program state might not be as intended.
3. **The infection propagates.**
   Or it can be overwritten, masked, or corrected
   by later program instructions.
4. **The infection causes a failure**, i.e., an observable error in the program behavior.

> Not every defect results in an infection
> and not every infection results in a failure.

Reasons for Defects

defects (bugs) are inherent parts of programs:

- mistake by the programmer
- incomplete/changing requirements
- incompatible interfaces of modules
- unpredictable interaction of multiple components in a distributed environment
- ...

Why does a program fail, and how can we fix it?

Verification & Validation vs. Debugging

verification and validation

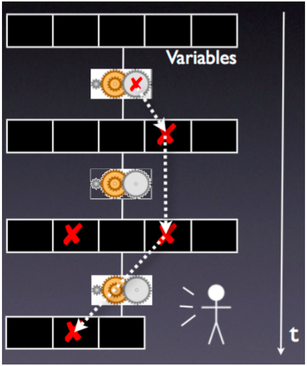
show existence of defects

- not every defect causes a failure
- testing can only show the presence of errors – not their absence

debugging

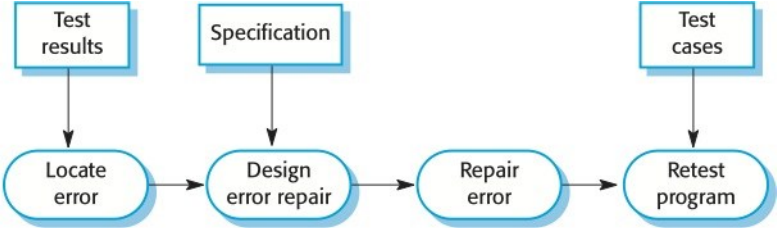
locate and correct defects

- relate failure to defect
- ... and remove it



from [Zeller09]

Debugging Process



from `http://iansommerville.com/software-engineering-book/web/debugging/`

7 Steps for Debugging – The Traffic Principle

Track the problem (bookkeeping in DB)

Reproduce the problem

Automate the simplification of the test case

Find origins possible infection origins

Focus on the most likely origins

Isolate the infection chain

Correct the defect

Complexity of Debugging

find a defect

=

isolate the transition from a sane state infected state

=

search in space and time

Complexity of Debugging

find a defect

=

isolate the transition from a sane state infected state

=

search in **space** and time

- **space**
  - a state consists of the current values of the variables of a program and a program counter
  - which part of a state has to be inspected to find an infection?

Complexity of Debugging

find a defect

=

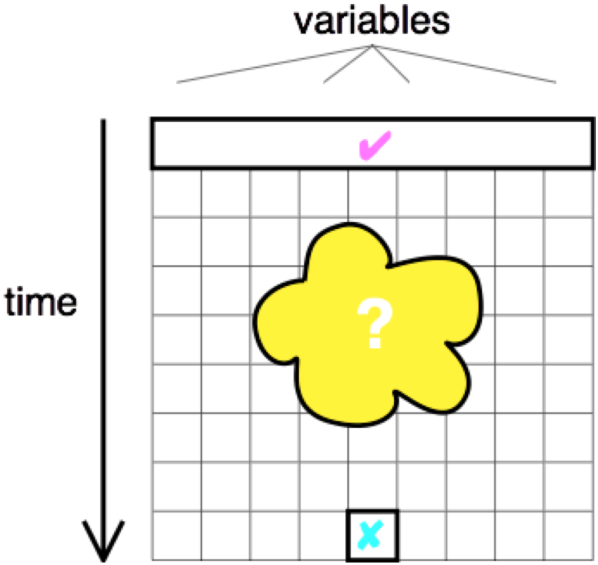
isolate the transition from a sane state infected state

=

search in space and **time**

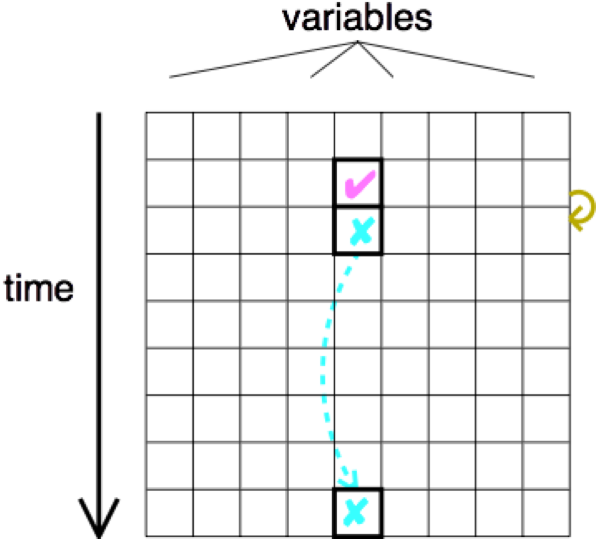
- space
  - a state consists of the current values of the variables of a program and a program counter
  - which part of a state has to be inspected to find an infection?
- **time**
  - the program execution consists of many states
  - when does the infection take place?

From Defects to Failures



from [Zeller09]

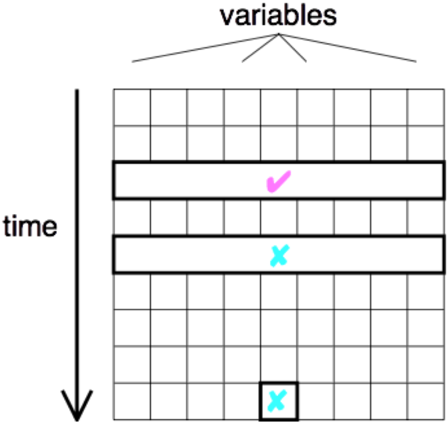
From Defects to Failures



from [Zeller09]

Basic Debugging Principle I

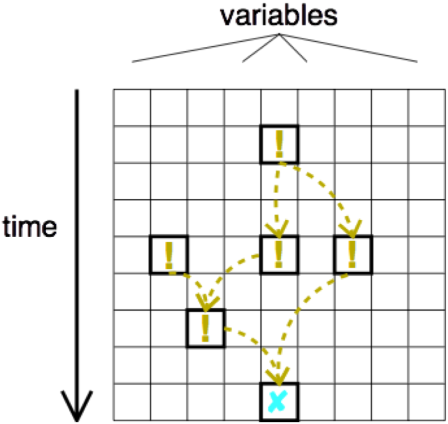
separate sane from infected



from [Zeller09]

Basic Debugging Principle II

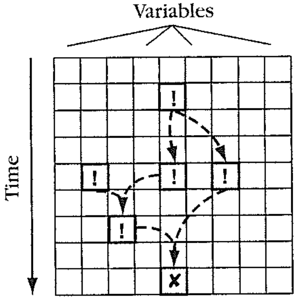
separate relevant from irrelevant



from [Zeller09]

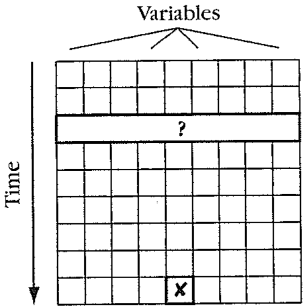
Techniques for Automating Debugging

- **program slicing**
- observing & watching of states
- asserting invariants
- detecting anomalies
- isolating cause-effect chains



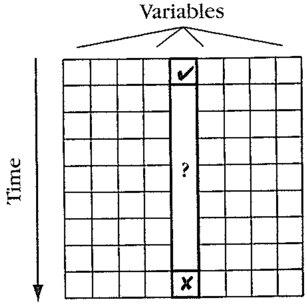
Techniques for Automating Debugging

- program slicing
- **observing** & watching of states
- asserting invariants
- detecting anomalies
- isolating cause-effect chains



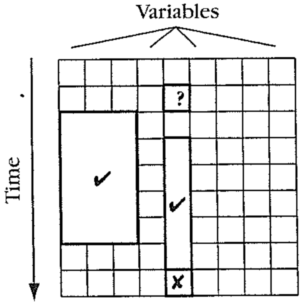
Techniques for Automating Debugging

- program slicing
- observing & **watching of states**
- asserting invariants
- detecting anomalies
- isolating cause-effect chains



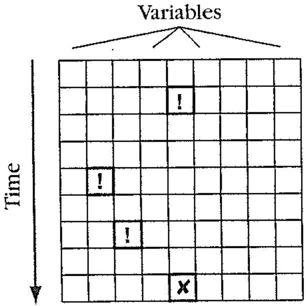
Techniques for Automating Debugging

- program slicing
- observing & watching of states
- **asserting invariants**
- detecting anomalies
- isolating cause-effect chains



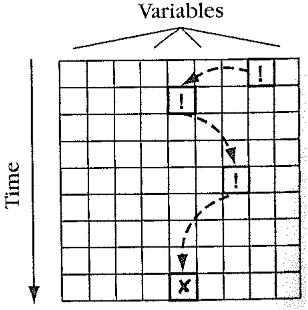
Techniques for Automating Debugging

- program slicing
- observing & watching of states
- asserting invariants
- **detecting anomalies**
- isolating cause-effect chains



Techniques for Automating Debugging

- program slicing
- observing & watching of states
- asserting invariants
- detecting anomalies
- **isolating cause-effect chains**



Example: Broken Shell-Short

```
void shell_sort (int a[], int size) { ... }

int main (int argc, char *argv[]) {
  int *a, i;

  a = (int *)malloc ((argc - 1) * sizeof(int));
  for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

  shell_sort (a, argc);

  printf ("Output: ");
  for (i = 0; i < argc - 1; i++) printf("%d ", a[i]);
  printf("\n");

  free(a);
  return 0;
}
```

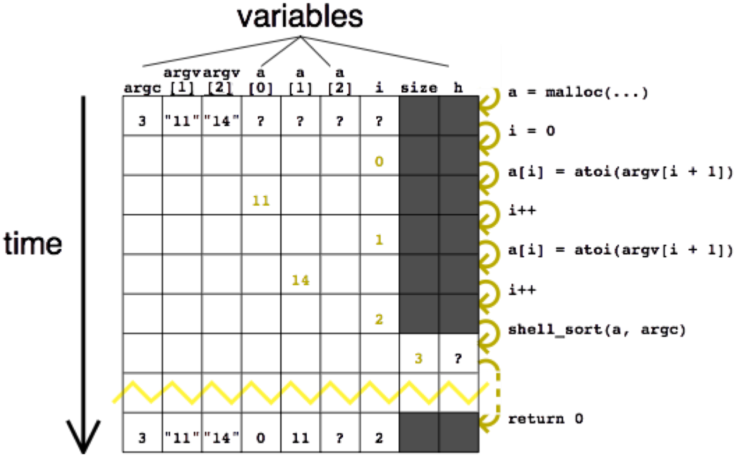
```
$ shell_sort 9 8 7
Output: 7 8 9

$ shell_sort 11 14
Output: 0 11
```

JKU

Demo: Debugging Broken Shell-Sort

- where does the “0” in `a[0]` come from?
- when does the infection happen?



from [Zeller09]

Common Crash Scenarios

1. Application works as expected and never crashes.
2. Application crashes due to rare bugs that nobody notices or cares about.
3. Application crashes due to a commonly encountered bug.
4. Application deadlocks and stops responding due to a common bug.
5. Application crashes long after the original bug.
6. Application causes data loss and/or corruption.

https://blog.codinghorror.com/whats-worse-than-crashing/

What's a Problem?

A problem is a questionable property of a program run. It becomes

- ... a failure if it's incorrect
- ... a request for enhancement if it is a missing feature
- ... a feature if it reflects normal behavior

Problem Life Cycle

- The user informs the vendor about some problem.
- The vendor
  1. reproduces the problem
  2. isolates the circumstances
  3. locates and fixes the defect
  4. delivers the fix to the user

Large Scale Debugging Processes

organizations of the problem life cycle:

- which problems are currently open?
- which are the most severe problems?
- did similar problems occur in the past?

management of problems requires more than a TODO list

Facts about the Problem

- problem history: how to reproduce the problem
  - accessed resources (input files, configurations)
  - circumstances necessary for the problem to occur
  - as simple as possible
- diagnostic information of the program
  - logging features of the program
  - stack traces of the operating system
- symptoms of the problem
- expected behavior

  bug or feature?

Facts about the Problem

- product release
- operating environment
- system resources

crucial if the problem depends on specific product or environment features

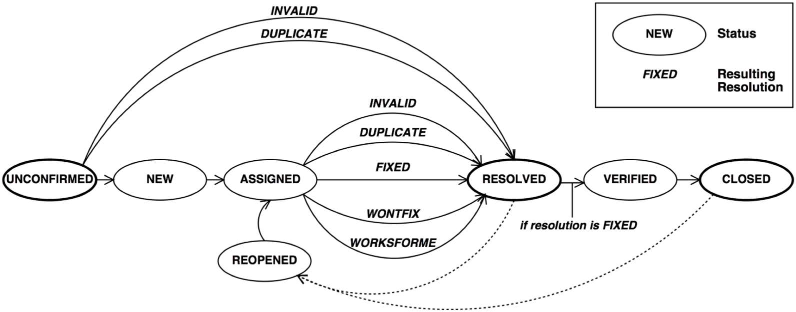
⇒ automatic collection possible

⇒ privacy issues from internal information like core dumps, log files about user actions, ...!

General Outline of a Bug Report

- summary
- component
- version
- operating system
- description
- steps to reproduce
- actual results
- expected results

Life cycle of a Bugzilla Bug



from [Zeller09]

Features of Issue-Tracking Systems

- severity classification
  - enhancement. A desired feature.
  - trivial. Cosmetic problem.
  - minor. Problem with easy workaround.
  - normal. “Standard” problem.
  - major. Major loss of function.
  - critical. Crashes, loss of data or memory
  - showstopper. Blocks development.
- priority
  - higher the priority, sooner to be addressed
  - independent from severity
- identifier
- comments
- notification

Responsibilities

Who ...

- ... files problem reports?
- ... classifies problem reports?
- ... sets priorities?
- ... takes care of the problem?
- ... closes the issue?

in many organizations: software change control board

Challenges

- as many facts as possible to reproduce the problem vs. as few facts as possible to find duplicates
- relate version of product with problem
  - binaries of user
  - all sources of specific releases
  - recreation of any given configuration

  $\Rightarrow$ tool support software configuration management like version control systems:
  - tag releases
  - storing of fixes in branches
- failing test cases should make bug reports obsolete