Software Reliability

Lecture 1

Introduction

Cristian Cadar & Alastair Donaldson

http://multicore.doc.ic.ac.uk/SoftwareReliability/

Team

Lecturers:

- Cristian Cadar
  - c.cadar@imperial.ac.uk
  - https://www.doc.ic.ac.uk/~cristic
- Alastair Donaldson
  - a.donaldson@imperial.ac.uk
  - https://www.doc.ic.ac.uk/~afd

Course Support Leader:

- Luis Pina
  - l.pina@imperial.ac.uk
  - https://www.luispina.me

Schedule

- 27 slots
  - 19 lectures
  - 5 tutorials
  - 1 tool demo
  - 1 coursework discussion
  - 1 guest lecture

Recording

We plan to record most lectures and make them available online

In the past this mainly, but not always, worked

Don’t rely on the lecture recordings: treat them as a bonus

Defects in software systems

Most software problems caused by programmer errors that could be avoided through better testing and verification

Famous examples include:

- **Therac-25** radiation therapy machine led to massive radiation overdoses. Many root causes, including **arithmetic overflow** & concurrency **race conditions**
- **Ariane 5** test flight: error caused by **data conversion** from 64-bit floating point value to 16-bit signed integer value led to self-destruction
- **Microsoft Zune** **termination bug**: infinite loop during date calculation – caused all devices to hang simultaneously! http://techcrunch.com/2008/12/31/zune-bug-explained-in-detail/

http://www.devtopics.com/20-famous-software-disasters/

Widely used approaches for improving software reliability

Manual testing: programmers construct test cases, either to achieve high **coverage**, or in response to known **bugs**

Coding standards, code review: developers conform to a set of coding standards, commits are subject to code review by colleagues. E.g.:

- **Joint Strike Fighter coding standards:**
  http://www.stroustrup.com/JSF-AV-rules.pdf
- **LLVM coding standards:**
  http://llvm.org/docs/CodingStandards.html

Tool support:

Debuggers (**gdb**), memory analysers (**Valgrind**), refactoring aids (**Eclipse**), testing frameworks (**JUnit**), bug trackers (**Bugzilla**)

Limitations of manual testing

No guarantees. A successful test **exposes a bug**, and can help ensure the bug does not return. No amount of testing guarantees that a system is free from defects

High manual effort. Writing tests to achieve high coverage is time consuming (thus expensive)

Limitations of human thought. Human testers do not tend to spot intricate, unusual input combinations that may lead to failure

Course focus: beyond manual testing

We will study:

Verification techniques, which aim to prove program correctness

Bug-hunting methods, which aim to reveal defects and generate corresponding tests

Two general methods:

Static analysis, which analyses the source code without running the program

Dynamic analysis, which analyses running programs

Changes based on SOLE

- Reduction in intensity of coursework
- Introduction of KLEE tool demo
- Course Support Leader (Luis) to help with tutorial sessions

Topics

Verification condition generation
Procedure summaries
Bounded model checking
Dynamic symbolic execution
Constraint solving
Invariant generation
Systematic testing for concurrent programs
The lockset algorithm
Undefined behaviour, compiler bugs and unstable code
Intro to security and stack canaries
Safe C compilers
Control-flow, data-flow and write integrity

Underpinning many of these techniques: **SMT solvers**

Reading research papers

Software reliability: an **extremely active research area**

Techniques we will cover are relatively new, still being actively investigated

No textbooks covering all the course topics

We recommend reading all research papers underlying the course

- The content of three papers is **directly examinable**
- Paper information posted on the course website as the course progresses

Examination

Written exam (67%)

- Answer 2 of 3 questions
- Exam questions draw on material from lectures, tutorial sheets, practical assignment and examinable papers

Coursework (33%)

- The coursework is one large assignment, building a program verifier
- Split into Part 1 (5%) and Part 2 (28%) to help you manage your time

Coursework

Warning: coursework intensive!

Your task is to implement a program verifier for Simple C, a C-like programming language

Part 1: build a verifier for loop-free, call-free programs.
Deadline: 28 October

Part 2: build a full-fledged verifier for multi-procedure programs with loops and calls.
Deadline: 25 November

Coursework: undertaken in groups of up to 3

Deadline for group formation: noon on 19 October

Should I take the course?

Software Reliability is a high workload course:

- The coursework is **demanding**, requiring a lot of technical programming; **it will be time-consuming**
- The three research papers are lengthy and technical, and we recommend reading additional papers

We'll be delighted if lots of you take the course, but **it is not an easy option**

Any questions on the course structure?

Details and updates on the course web pages:

http://multicore.doc.ic.ac.uk/SoftwareReliability/

Preliminaries

- Bugs and correctness
- General vs. functional properties
- Safety (and liveness) properties
- Static and dynamic analysis
- False positives and false negatives

Bugs in Software

Out-of-bounds array access

Incorrect algorithm

Invalid dynamic cast

Mistake in implementation of algorithm

Division or modulo by zero

Integer overflow

Infinite loop

Double free

Access after free

Unbounded recursion

Null pointer dereference

Assertion failure

Deadlock

Data race

…and MANY more

Classifying Software Bugs

General/generic bugs

Division or mod by zero
Assertion failure
Integer overflow
Invalid dynamic cast

Memory bugs

Out-of-bounds access
Double free
Null pointer dereference
Access after free …

Concurrency bugs

Deadlock
Data race …

Termination bugs

Infinite loop
Unbounded recursion

Functional bugs

Incorrect algorithm
Mistake in implementation of algorithm

(not a precise or complete classification)

Functional properties vs. general properties

General properties: **we know what to check!**

E.g., array bounds checking:

When we see **A[e]** we must assert **e** is in range

…might be hard (e.g., in C) to know exactly what the range is, but at least we know what range **means**

…but a system can be free from general defects and yet behave nonsensically!

Functional properties vs. general properties

Functional properties: don't know *a priori* what to check

Is this method correct?

```
// Precondition: A points to an allocated array
// of n integers, n > 0
int findSmallest(int *A, int n) {
  int min = A[0];
  for(int i = 1; i < n; i++)
    if(A[i] > min) min = A[i];
  return min;
}
```

From a software reliability tool's perspective, **yes!** How does the tool know what the programmer wanted?

Functional properties vs. general properties

Checking functional properties requires *specifications*

Specs can be hard and laborious to write, and require maintenance

```
int findSmallest(int *A, int n)
  requires \allocated(A, int, n),
  requires n > 0,
  ensures (\forall int x .
      0 <= x && x < n ==> \result <= A[x])
{
  int min = A[0];
  for(int i = 1; i < n; i++)
    if(A[i] > min) min = A[i];
  return min;
}
```

This is a very simple spec, and it is not even complete (why?)

Think about how you would write a spec to say that a binary tree is balanced…

Functional properties vs. general properties

Functional verification: are the benefits worth the effort?

- For safety critical code: **yes**
- In general: at present, **no** (but still a fascinating topic!)

General properties often **easier to automatically check**:

- Functional properties often involve quantifying over all elements of a data structure, general properties often do not
- Quantifiers present a challenge for automated theorem provers and constraint solvers

Result: most software reliability tools focus on analysis of general properties

Philosophy: let's help developers weed out the general defects first, allowing them to concentrate on functionality

Safety properties: the focus of this course

Safety properties – “something bad does not happen” – can be expressed using *assertions* + program instrumentation

- **No null dereferences:** insert assertion before each dereference
- **No out-of-bounds accesses:** insert assertion before each array access + (depending on language) instrumentation to keep track of ranges and object referents
- **No divisions by zero:** insert assertion before each division
- Methods **init**, **process** and **destroy** may only be called in sequence: add instrumentation *variables* to track order of calls, and assertions over these variables

When building analysis tools we can restrict attention to assertion checking

Beyond the scope of this course

Liveness properties, e.g.: *“is every packet received eventually acknowledged”*, or *“does the algorithm eventually halt for every well-formed input”*

Non-functional properties, e.g. related to performance, memory usage or energy consumption

Nevertheless, variants of many of the techniques we cover can be applied for analysis of liveness and non-functional properties

Dynamic analysis

Involves running programs (either directly or through emulation) and collecting information about executions

Advantages:

- Precise (only observe what the program can actually do)
- Scalable (in many cases proportional to regular execution)

Disadvantages:

- Requires whole system (hard to dynamically analyse a method in isolation, need a test driver)
- Requires execution environment or simulator
- Usefulness of result determined by quality of test inputs

Dynamic analysis

What are the most popular kinds of dynamic analyses?
Widely used dynamic analysis techniques include:

Valgrind (memory error detection and more)
Compiler sanitizers, e.g, **ThreadSanitizer**
(detecting concurrency errors)

We will study several dynamic analysis techniques (which also incorporate static analysis components):

- **dynamic symbolic execution**
- **systematic testing for concurrent programs**
- **lockset algorithm**
- **compiler fuzzing**
- **stack canaries**
- **control-flow, data-flow and write integrity**

Static analysis

Reasoning about program executions without actually running the program

Advantages:

- Can detect defects not revealed by existing test cases
- High coverage: can potentially prove properties about all, or a large number, of possible executions
- Can be applied to incomplete systems – applicable at early stages of development
- Potentially highly scalable if applied in a **modular** fashion

Key disadvantage: static analysis ranges between

- Precise, but extremely expensive
- Fast, but extremely imprecise (lots of **false positives**)

Static analysis

What are the most popular static analyses?

Widely used static analysis techniques:

- **Compilers** (think of the warnings they generate)
- Open-source tools, e.g., **lint** (C), **FindBugs** (Java)
- Commercial tools from e.g., **Coverity** and **GrammaTech**
- Internal company tools, e.g., **Facebook's Infer** tool

We will study:

- verification condition generation
- procedure summaries
- bounded model checking
- invariant generation

False positives

An analysis is said to report a **false positive** if it warns about a problem that **cannot actually occur**

Term may seem counter-intuitive, because **positive** sounds good!

Easy way to remember: a **bug** is like a **disease**

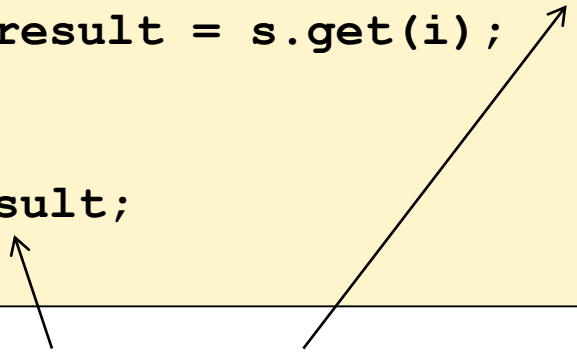
- You test **positive** for a **disease** if you have the disease, which is **bad**
- A **false positive**: you are told you have a disease, but really you do not

An analysis is said to be **imprecise** if it reports lots of false positives

An analysis that may report false positives is sometimes called **incomplete**

False positive: example

```
int findSmallest(List<Integer> s) {
    if(s.isEmpty()) throw new RuntimeException();
    int result;
    for (int i = 0; i < s.size(); i++) {
        if(i == 0) {
            result = s.get(i);
        } else if(s.get(i) > result) {
            result = s.get(i);
        }
    }
    return result;
}
```



error: variable result might not have been initialized

…but is this a useful error message?

False negatives

An analysis is said to report a **false negative** if it reports **absence** of problems, when actually problems **can occur**

Easy way to remember: a **bug** is like a **disease**

- You test **negative** for a **disease** if you **do not** have the disease, which is **good**
- A **false negative**: you are told you do not have a disease, but really you do!

An analyser is called **unsound** if it may report false negatives

False negative: example

```
int inc(int x)
  ensures \result > x
{
  return x + 1;
}
```

Post-condition

Does the post-condition hold for all inputs?

What about **`Integer.MAX_VALUE`?**

Some analysers do not warn about this problem, treating integers **mathematically**. Strictly this is **unsound**

… but … warning about overflow here could be regarded as a *false positive*, if the method is required to be called with appropriately small arguments

False positives/negatives are not absolute – may depend on context

False positives vs. false negatives

False negatives are bad when analysing safety critical software – missing a bug can be disastrous

False positives hinder the use of analysis tools in day-to-day software development:

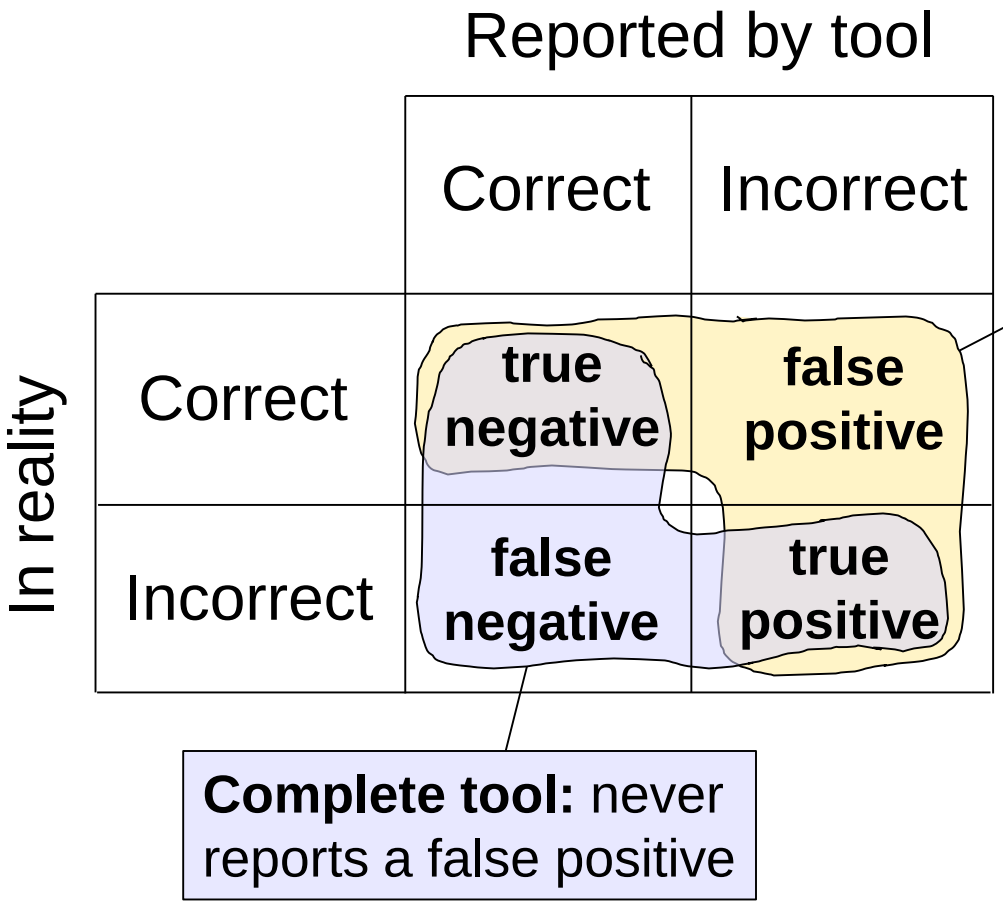
- If tool gives 90% false alarms, programmers tend to ignore **all** warnings

Commercial static analysers: **false positives regarded as the main problem**

- Unsoundness (false negatives) carefully introduced to limit false positive rate
- See Coverity paper: "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World" – Beesey et al., Communications of the ACM, Vol. 53 No. 2, Pages 66-75

Aid in remembering false positives, negatives, soundness and completeness

Compare tool's view of a program's correctness with reality



Important: in general, the tool has no idea whether its result is a true/false positive/negative