Software Verification

CS 6110, taught by Pavel Panchekha

Pavel Panchekha

T/R 14:00-15:30

WEB L126

Software Verification

No Bugs

Bugs are bad

Bugs are bad

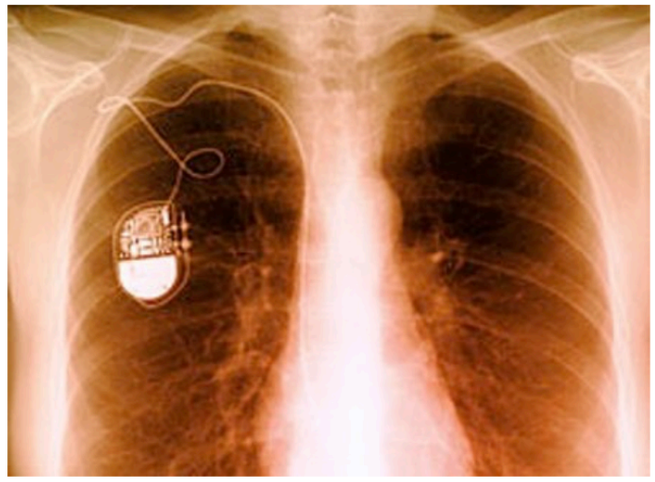
Software bugs are **dangerous and costly**

Security bugs, down-time, crashes, lost productivity

$59,000,000,000 annual cost (NIST estimate)

In 2002!

Bugs are bad



Therac 25

Radiation therapy software

Atomic Energy of Canada

Race condition **kills 5**

ION Therapy Planning

Dosage calculation for radio-therapy

Panamanian health organization

8 killed, 20 injured, FDA investigation

Ariane 5

Ariane 4 rocket controller reused

Faster rocket causes integer overflow

Flight computer crashes 40 seconds in

2003 BMW Crash

Thai Finance Minister **trapped in car**

Computer crashes, doors stuck

Firemen **smashed windows**

2005 Prius Recall

Cars **stall, stop at highway speeds**

Software bug causes computer crash

75,000 cars recalled

What is a Bug?

Feature request

Confusing features

Ugly, unintuitive

Unusably slow, memory hog

Confusing error messages

Unhandled exceptions

Segmentation fault

What is a Bug?

Feature request

Confusing features

Specification

Unusable slow, memory hog

A bug is a mismatch between code and its specification

Confusing error messages for crashes

Unhandled exceptions

Segmentation fault

Specifications

Exact description of what a program should do

What it does, not how it does it

`quicksort(l)` // Spec: **sorted(output)**

Intermediate specifications

What needs to be true of **`partition`** for **`quicksort`** to work?

Program Reasoning

Spec: what a program **should** do—what **does** it do?

Matching program execution to clauses in spec

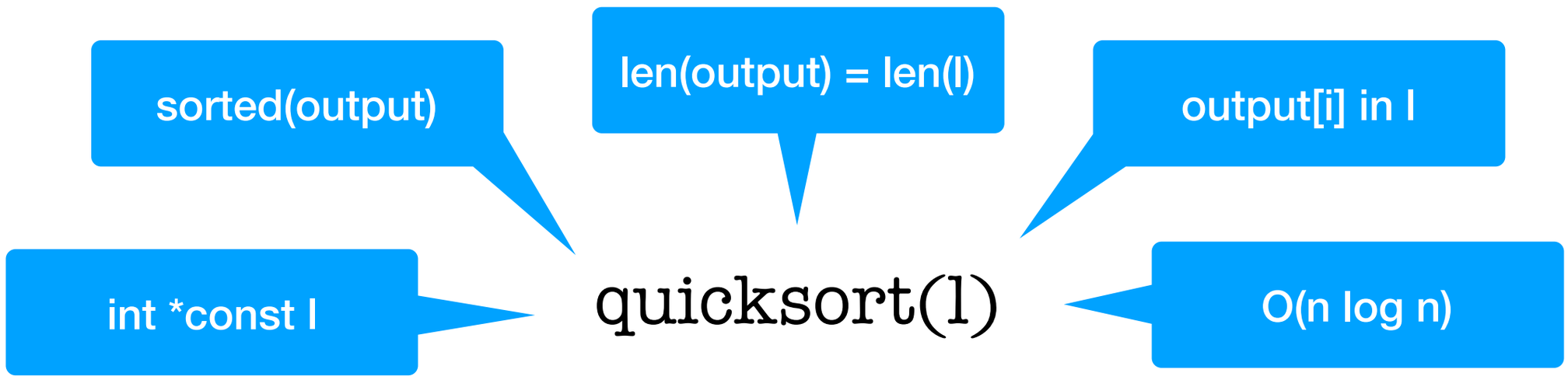
```
// len(l) == 3 and l[0] == 5
quicksort(l)
// len(l) == 3 and ???
```

Program reasoning is running the program “**in your head**”

You can run it **backwards or forwards**

Scaling up

How do you choose **which specs to prove**?



Choose the specificity of your specification

Fine-grained or coarse-grained specifications can be best

About this Course

Lectures, Assignments, Projects, Grading

Your Instructor

Pavel Panchekha

PhD in 2019

U. of Washington

I'm new here!

I work on **the browser as a platform** for executing programs with PL techniques

Your TA

Manasij Mukherjee

PhD student

Advisor: John Regehr

Works on compiler optimization using **program synthesis** and **static analysis**.

Three Parts

Write a specification

Propagate into program

Scale to systems

Three Parts

Logical reasoning

Program logics

Static analysis

Six Topics

Logical reasoning

First-order logic

Deciding truth

Program logics

Symbolic execution

Axiomatic reasoning

Static analysis

Abstract domains

Dependent types

Six Assignments

Logical reasoning	Program logics	Static analysis
First-order logic	Symbolic execution	Abstract domains
N-Queens with miniSAT	**Bug-finding with KLEE**	**Analysis with Checkers**
Deciding truth	Axiomatic reasoning	~~Dependent types~~
Kenken with Z3	**Verification with Dafny**	

Six Assignments

Assignments **every two weeks**ish

Due Thursdays at midnight

Late policy: 20 point deduction without 48hr notice

Each assignment **10% of your final grade**

No assignment for dependent types because **projects**

Course Project

Verify something, build a verifier, contribute, etc…

Proposal Draft	Early February
Proposal Presentation	Mid-February
Milestone 1	Early March
Milestone 2	Late March
Final presentation	Mid April

Project is **40% of your final grade**

Question: Team projects? Alternative assignment?

Participation

10% of your final grade, half **attendance** half **feedback**

24hr notice for absence from class (email is fine)

Feedback form mandatory **every class**

Answers aren't graded, I'll pretend they're anonymous

A few standard questions plus **class polls**

Web Pages

canvas

piazza

my.eng.utah.edu/**~cs6110**

pavpan@cs.utah.edu

manasij@cs.utah.edu

How to Verify

From specification to verification

Quicksort

```
def quicksort(l):

    pivot = l[len(l)//2]

    left, right = partition(l, pivot)

    left2 = quicksort(left)

    right2 = quicksort(right)

    return left2 + right2
```

Quicksort

Post: sorted(**output**)

sorted(l) := **l[i] ≤ l[i+1]**

```
def quicksort(l):

    pivot = l[len(l)//2]

    left, right = partition(l, pivot)

    left2 = quicksort(left)

    right2 = quicksort(right)

    return left2 + right2
```

Quicksort

Post: sorted(**output**)

```
def quicksort(l):

    pivot = l[len(l)//2]

    left, right = partition(l, pivot)

    left2 = quicksort(left)

    right2 = quicksort(right)

    return left2 + right2
```

Prove: left[i] < right[j]

← sorted(**left2**)

← sorted(**right2**)

?

Quicksort

Post: sorted(**output**)

```
def quicksort(l):

    pivot = l[len(l)//2]

    left, right = partition(l, pivot)   ← left[i] < right[j]

    left2 = quicksort(left)   ← sorted(left2)

    right2 = quicksort(right)   ← sorted(right2)

    return left2 + right2   ← sorted(left2 + right2)
```

Nope! Is left2[i] < right2[j], or just left and right?

Quicksort

Post: sorted(**output**) and **output[i] in l**

```
def quicksort(l):

    pivot = l[len(l)//2]

    left, right = partition(l, pivot)   ← left[i] < right[j]

    left2 = quicksort(left)   ← sorted(left2)

    right2 = quicksort(right)   ← sorted(right2)

    return left2 + right2   ← sorted(left2 + right2)
```

Prove: left[i] in l and right[j] in l

Quicksort

Post: sorted(**output**) and **output[i] in l**

```
def quicksort(l):

    pivot = l[len(l)//2]

    left, right = partition(l, pivot)   ← left[i] < right[j] and left[i] in l and right[j] in l

    left2 = quicksort(left)   ← sorted(left2) and left2[i] in l

    right2 = quicksort(right)   ← sorted(right2) and …

    return left2 + right2   ← sorted(left2 + right2) and …
```

The right specification?

Quicksort

Post: sorted(**output**) and **output[i] in l** and **l[i] in output**

```
def quicksort(l):

    return []
```

Nope!

Quicksort

Post: sorted(**output**) and **output[i] in l** and **l[i] in output**
and **len(l) = len(output)**

```
def quicksort(l):

    return [min(l)] + sort(l)
```

Nope!

Quicksort

Post: sorted(**output**) and **output[i] in l** and **l[i] in output**
and **len(l) = len(output)**

```
def quicksort(l):

    if l == [1, 2, 3, 3, 4]:
        return [1, 2, 3, 4, 4]
    else:
        return sort(l)
```

Nope!

Quicksort

Post: sorted(**output**) and **l.count(x) == output.count(x)**

```
def quicksort(l):

    if l == [1, 2, 3, 3, 4]:
        return [1, 2, 3, 4, 4]
    else:
        return sort(l)
```

Is that finally enough?

Challenges

Writing a specification for `quicksort`

Reasoning about predicates like sorted

Combining facts about lists and predicates

Propagating facts through the program

Expanding the specification to make it provable

Verification is hard!

Recent Successes

Quark

COMPCERT

Memory Model

IronClad

ROSETTE

Z3

SLAM

TLA+

KODKOD

Dafny

seL4

frenetic >>

alloy

SAGE

Static analysis based fuzzing for Windows applications

500+ years of machine time

Hundreds of security issues identified

One in three bugs in Win7 WEX identified by SAGE

Millions of dollars saved

Verify **drivers don't crash** Windows

> … in some very key areas, for example, driver verification we're building tools that can do actual proof about the software …
>
> — **Bill Gates**, WinHec 2002 Keynote

Standard part of driver SDK

Prevented **hundreds of blue screen bugs**

IronFleet

Verified implementation of **distributed system**

Provably **resistant to hardware failure**

Found **bugs in standard algorithms**

Next class:

Boolean Logic

To do:

- ☐ Course feedback
- ☐ Register for Piazza
- ☐ Find course webpage