EECS498-003
Formal Verification of Systems Software

Material and slides created by

Jon Howell and Manos Kapritsos

About me

Manos Kapritsos (manosk@umich.edu)

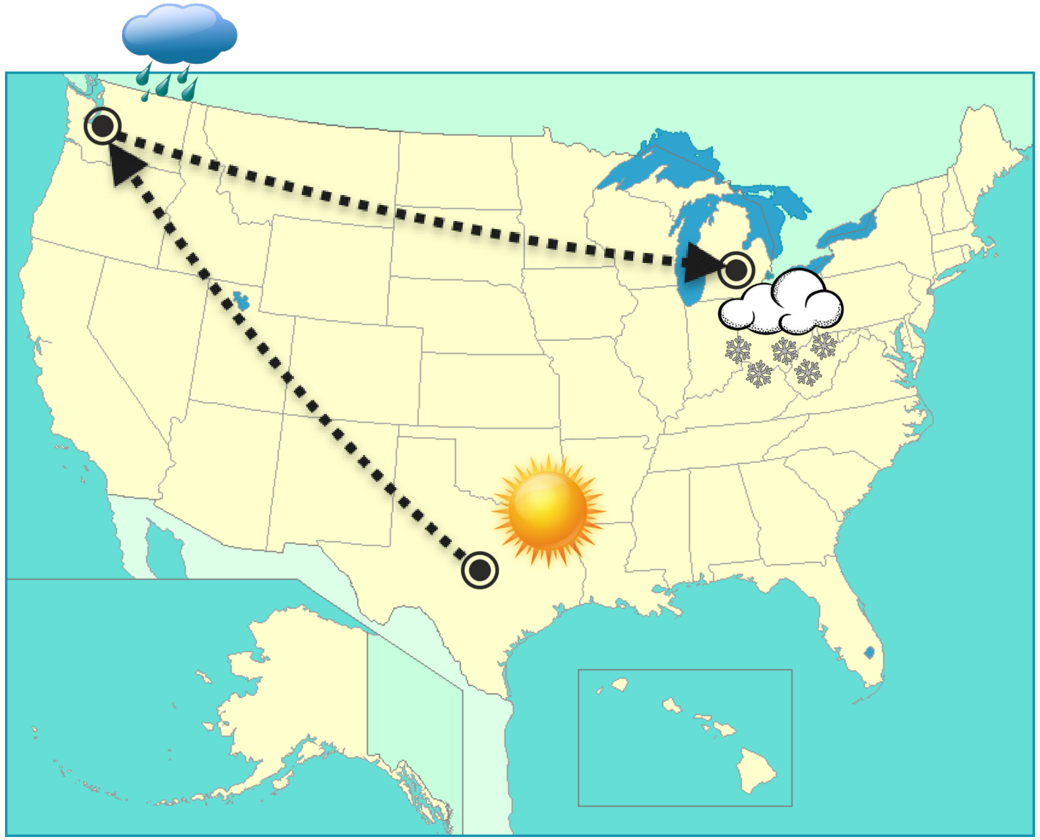
Areas of research: Formal Verification, Distributed Systems

29+1 years old (but I am not 30)

Disclaimer

Any resemblance to purple-looking super villains, living or dead, is purely coincidental

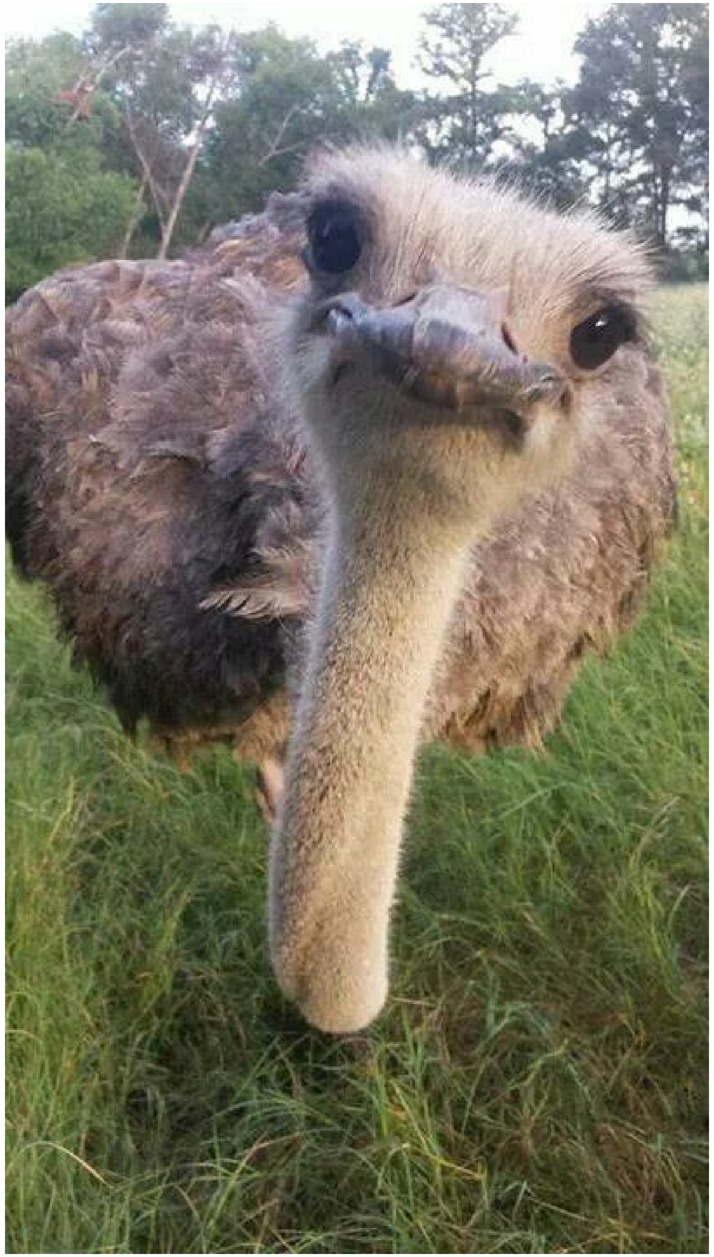
About me



About you and me

- I love teaching and interacting with my students

- I want to get to know you all by name



It’s picture time!

1. Give your phone to someone close to you
2. Pose for the picture
3. Go to verification.eecs.umich.edu/self.php (login may be required)
4. Upload your picture!
5. (optional) update your preferred name

About you and me

- I love teaching and interacting with my students
- I want to get to know you all by name
- I'm here to help. Come to me with any question!
  - course-related: office hours
    - Thursday, 11am-12pm, BBB 4824, starting next week
  - Life, The University, and Everything: any time

If you need more help...

Our GSI, Keshav Singh, is here to help

A Jedi Master in Formal Verification

Office hours: Monday 4:30-5:30, Learning Center Table #1

The unseen hero

Jon Howell, VMWare Research

“Bugs, unit tests, gdb. The dark side are they.”

“Testing not make one great.”

Co-designer of the material in this course

Agenda for Today

- Why learn formal verification?

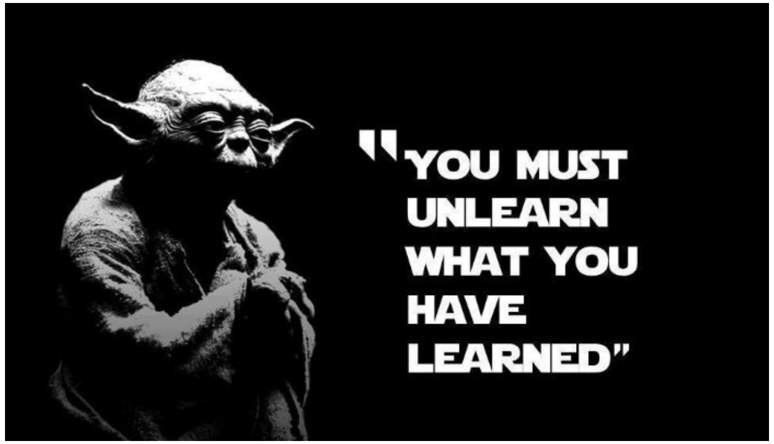
- Course syllabus and logistics

- What is formal verification?
  - (and other, related approaches)

- Getting started with Dafny

Why learn formal verification?

EECS280/281	High-level programming
EECS482	OS-level programming
EECS491	Distributed programming



We have taught you how to write and test programs

Real-world systems are too complex to test

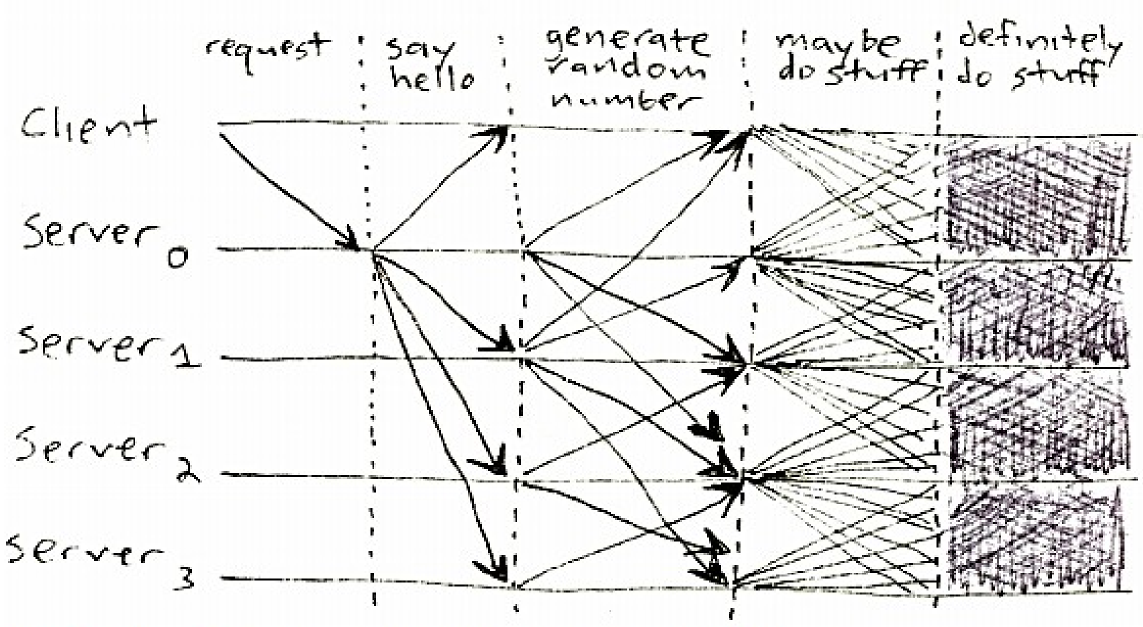


Figure 1: Typical Figure 2 from Byzantine fault paper: Our network protocol

[Mickens 2013]

Despite tremendous effort…

"not one of the properties claimed invariant in [PODC] is actually invariantly true of it."

[Zave 2015]

Under the same assumptions ade in the Chord papers, the [SIGCOMM] version of the rotocol is not correct, and not one of the properties claimed invariant in [PODC] is actually invariantly true of it. The [PODC] version satisfies one invariant, but is still not correct. The results are presented by means of counterexamples to the invariants in Section 4. In preparation for the results, Section 2 gives a brief summary of the protocol and failure assumptions, and Section 3 introduces the invariants.

Is it possible to write code that is completely bug-free?

Formal verification is as close as we can get

March 25, 2015

(or how I became a believer)

- One day before the deadline of SOSP’15, we had not yet run our code
- …and yet our code ran correctly the first time we ran it!
  - (and every time afterwards)

Verification in the real world

- Verification is an increasingly popular approach

- Many companies have picked it up in the last few years:
  - Microsoft, Amazon, Facebook
  - NASA, Boeing
  - Many blockchain/fintech companies

How will you benefit from this class?

- It will make you a better programmer
  - Whether you end up writing verified code or not
- You will learn to specify your code
  - To express your intent clearly and unambiguously
  - Just like design docs, but better
- You will learn important concepts
  - E.g. inductive invariants, refinement
- Get ahead of the curve
  - Learn an emerging skill

Testimonial from last semester

"I had a ~30 minute coding interview (in Python) and I **didn't have a single runtime or compilation [error]** after writing multiple class definitions ~100 lines of code. I have become a **better programmer** and approach problems in a **more structured way** because of this class and it was a large reason I got the position (as my interviewer has since told me)."

Anonymous student, Winter 2024

Objectives of this class

- Understand the fundamentals of formal verification
- Learn how they apply in a (distributed) systems context
- Get hands-on experience with proving systems correct
- Become familiar with a practical verification language

Prerequisites

- Experience with programming
  - i.e. EECS281
- I will explain the mechanics of any distributed systems in the class for those of you that haven't taken EECS491
- *No verification experience required*

If anything is unclear, do not hesitate to ask

About this class

- Disclaimer: this class is not formally verified! 
- This is the third time I am teaching this class
  - …and the first class ever on this material
- There is no textbook (anywhere!)
  - Jon and I are currently writing one (based in part on the experience from this class)

Class material

- Class webpage
  - https://verification.eecs.umich.edu
- Syllabus, lecture slides, problem sets and projects will be posted on the class webpage
- Subscribe yourself to Piazza
  - Announcements and class discussion

Enrollment

- We should be able to accommodate up to 50 people
- If you cannot enroll in the class for some reason, come talk to me

Lectures

- Lectures will be held in person in DOW 2166
- Recordings will be posted at: https://leccap.engin.umich.edu/leccap/site/z15i8cyn7j0y6wsp2vm
- (link is also on course web page and Piazza)
- I will be posting slides of each lecture (shortly) before the lecture, in case you want to keep notes directly on the slides

Lectures schedule

- Material is divided into six chapters, plus some advanced topics
  - Chapters 1-4 (before midterm) cover the basic concepts of verification
    - Basic verification and Dafny mechanics
    - Specification
    - Centralized state machines
    - Proving properties and inductive invariants
  - Chapters 5-6 (after midterm) cover distributed systems and refinement proofs
    - Distributed state machines
    - Refinement
    - Advanced topics

Problem sets and projects

- There are four (programming) problem sets and two projects
  - Problem sets will be done **individually**
  - Projects will be done in **groups of 1-2** students

- All deliverables will be submitted via the autograder.io website
  - They use a combination of auto-grading and hand-grading

Class workload

```
var difficulty
if student.degree == undergraduate {
     difficulty := “light to medium”
} else {
     difficulty := “heavy”
}
print “This class is “+difficulty
```

Policies

- Submission
  - Three submissions per day to the autograder
  - Due at midnight on deadline
  - Three late days throughout the semester

- Collaboration
  - Okay to clarify problem or discuss Dafny syntax
  - Not okay to discuss solutions

Exams

- Midterm: October 17, 6-8pm
- Final: December 18, 8-10am (sorry, the registrar sets this time!)

- No makeup exams
  - Except in dire circumstances
  - Make sure you schedule your interviews appropriately

Grading breakdown

- Problem sets: 26%
  - PS1: 8% (Chapters 1 and 2)
  - PS2: 6% (Chapters 3 and 4)
  - PS3: 6% (Chapter 5)
  - PS4: 6% (Chapter 6)
- Projects: 30%
  - Project 1: 15%
  - Project 2: 15%
- Midterm exam: 22%
- Final exam: 22%

What is formal verification?

- Step 1: Specify the correctness of the system formally
- Step 2: Prove that the implementation conforms to the spec

If the spec expresses your correctness property, then your system is correct, subject to any assumptions you have made during your proof

Other approaches

Testing: run the system with a large and/or representative set of inputs to determine if it behaves correctly

- Quality depends on acumen of test designer
- Infeasible to achieve complete coverage for complex systems

Model checking: Model the system and ensure all possible states are safe

- Correctness depends on how accurate the model is
- Does not scale well to complex systems, especially those with infinite state spaces, like distributed systems

Statically checking for correctness

What we want is a “static correctness check”, akin to a static type check

You write your code normally, but if you introduce bugs the checker will tell you

When the checker complains, you have to spend some time to convince it that your code is right---if indeed it is

Using a Theorem Prover

Express the execution of the system and its correctness as a mathematical formula (done automatically by the language)

Give the formula to a theorem prover, effectively asking:
”If the system behaves this way, is it possible for its correctness to be violated?”

A negative answer means the system is proven to be correct
A positive answer means there is still work to do, either:

- the system is indeed incorrect
- the proof is incomplete

Using Dafny

- We will be using Dafny as our verification language
- Dafny is an imperative language designed with formal verification in mind
  - …and plenty of functional language features
- Dafny uses an SMT solver (Z3) to automate verification to a large degree
  - …but it needs our help sometimes
- Most of the high-level skills are transferrable…
  - …but some are specific to Dafny and/or automation

Getting started with Dafny

- In the lab on Friday, Keshav will go over instructions for installing Dafny 4.4

- The simplest way to use Dafny is via the Visual Studio plugin
  - Gives you a nice interface

- You can also invoke Dafny on the command line:
  - dafny myFile.dfy

Dafny in Docker

- We provide you with a Docker container that has Dafny pre-installed
  - Makes it easy to get started
  - Ensures everyone is using the same Dafny version as the autograder
  - Not highly recommended for the bulk of your development
- Download and run it like this:
  - `docker pull ekaprits/eecs498-009:latest`
  - `docker container run --mount src=$PWD,target=/home/autograder/working_dir,type=bind,readonly -t -i ekaprits/eecs498-009:latest`
- CAEN machines have some partial support for Docker
  - If you don’t have access to a machine that can run Docker, contact me ASAP

Things to do

- Browse the course web page
- Subscribe to Piazza
- Install Dafny 4.4 (during lab)
  - If that doesn't work: install Docker, pull image, run image