CpSc 513: Course Overview

Mark Greenstreet

January 7, 2020

Outline:

- What is verification?
- A simple example: binary search
- Course mechanics
  - You will survive.
  - You'll even have fun.

What is verification?

- We use mathematical methods, mostly mathematical logic, to show that a hardware, software, or cyber-physical design has some desired properties.
  - Often, this is seen as “finding bugs.”
  - Bug finding can be very important from a safety and/or cost point of view.
  - Formal methods also allow us to build more highly optimized designs:
- The kinds of techniques that we use:
  - Boolean satisfiability (SAT)
  - SAT augmented with decision procedures for other domains (SMT)
  - Reachability computation: model checking
  - Abstraction, approximation, refinement, and theorem proving

Who Cares?

The early days:

- Floyd & Hoare, and Dijkstra devised program-calculi to reason about algorithms.
  - Software modeled as mathematical objects. You can **prove** properties of software.
  - Largely seen as too hard, not-scalable, impractical.
- Attempts to automate verification
  - Cooperating decision procedures, e.g: Boyer & Moore, Nelson & Oppen, Shostak.
  - Model checking: Clarke, Emerson, Sifakis, Dill, McMillan, ...

Mainly an academic topic until ...

Who Cares?

Hardware Verification

- ~1992: Model checking practical for cache-coherence protocols.
  - Now, **every** cache protocol in major CPUs is model-checked.
  - And network protocols, and on-chip networks, and just about anything that can be modeled as finite-state machines.
- ~1995: the FDIV bug.
  - An error in the divider unit in the Intel Pentium processor.
  - Intel lost about $500M due to the error.
  - Now, **every** floating point unit in major CPUs is formally verified.
- Also in the 1990s: logical equivalence checking
  - Hardware synthesis tools (i.e. compilers) optimize to the edge of being wrong.
  - Occasionally, they optimize beyond that edge.
  - Development cycles are long, and bugs are expensive: formally check the synthesis output.
  - Also used for assertion checking.

Since the late 1990s and early 2000s, **formal verification is**

Who Cares?

Software Verification

- Lags hardware verification by about 20 years $\rightarrow$ **this is a very exciting time for verification!**.
- Currently used in real-world software development: Amazon, IBM, Microsoft, . . .
  - Check that low-level systems code follows kernel protocols.
  - Termination checks for device drivers.
  - Array bounds checks.
  - Find security exploits.
  - Check cloud services configurations for security flaws.

Who Cares?

Robots and Analog chip designers

- Provable safety for autonomous vehicles, medical devices, etc.
- Provably safety monitors for learning-based controllers.
- Complements traditional controller with “complete” methods for characterizing behaviour.
- Similar remarks apply to analog and mixed-signal circuit designs.

The Big Picture

	Theory, algorithms, & technology	Applications & examples
propositional ~5 weeks	SAT and SMT intro Symbolic Execution The DPLL algorithm Binary Decision Diagrams	Automatic Exploit Generation Automatic Test Generation Digital circuit equivalence checking
temporal ~5 weeks	Model checking Boolean program abstraction How SMT works Temporal logic	Microprocessor verification (Intel) Cache protocol verification Software model checking (Microsoft) More software analysis tools Verifying fault tolerance (Amazon)
continuous ~3 weeks	Linear Differential Inclusions Interval arithmetic Projectagons	Cyber–physical systems: autonomous vehicles Circuit verification State–of–the–art SW verification

Advanced topics:

Interpolants, bounded model checking (BMC), Abstraction and refinement (e.g. CEGAR), Statistical model checking, concolic execution, interactive theorem proving.

We probably won't cover all of these, but I'll give half–hour intro and overviews along with "for further reading" papers.

Binary Search

- This example is from "Programming Pearls: Writing correct programs", J.L. Bentley, *CACM*, Vol. 26 No. 12 (Dec. 1983), pages 1040-1045. (non-UBC link).
- Bentley had given courses for professional programmers at Bell (now AT&T) and IBM.
- Given an hour to solve the problem, $\sim$ 90% of programmers produced code that failed on a small set of test cases.
- Bently wrote:
  - *"I found this amazing: only about 10 percent of professional programmers were able to get this small program right."*
  - *"This exercise displays many strengths of program verification: the problem is important and requires careful code, the development of the program is guided by verification ideas, and the analysis of correctness employs general tools."*

Course mechanics

- We'll cover roughly one paper per lecture.
  - The reading list is at
    http://www.cs.ubc.ca/~mrg/cs513/2019-2/reading/ReadingList.html
  - Starting Jan. 9, you will be expected to write a short summaries for the papers. See slide 11.
- Each person will present one paper: see slide 12.
- There will be 4 ± 1 homework assignments: see slide 14.
- There will be a project: see slide 15.
  - Should take $\sim$ 40 hours of your time.
  - The final output is a M.Sc. thesis proposal (or equivalent). They are fine choices for papers to present.

Grading

$$\max\left(\frac{SummaryTotal}{20}, 1\right) * (0.4 * Homework + 0.4 * project + 0.4 * prese$$

- Paper summaries.
  - Worth $\frac{5}{4}$ points if turned in by noon the day the paper is covered.
  - Worth 1 point if turned in by 3:30 the day the paper is covered.
  - Submit your summary by sending e-mail to mrg@cs.ubc.ca
  - Plain ascii is prefered, PDF is acceptable – anything else may lose some points for annoying the instructor.
- 15% class presentation.
- 30% homework
- 45% course project.

Paper summaries

- Each summary should address each of the questions below:
  1. What problem does the paper address?
  2. What is the key insight/idea in the paper's solution to this problem?
  3. What did the authors do to demonstrate their claims? (e.g. implement a tool, present a proof, run simulations, etc.)
  4. Is the support for the claims convincing? Why or why not?
  5. What are your questions or other comments about the paper?
- If you found a paper too hard to read:
  - Write a description of where you got stuck.
  - Write some questions that would help you understand the paper.
- Grading:
  - Worth $\frac{5}{4}$ points if turned in by noon the day the paper is covered.
  - Worth 1 point if turned in by 3:30 the day the paper is covered.
  - Submit your summary by sending e-mail to mrg@cs.ubc.ca
  - Plain ascii is prefered, PDF is acceptable – anything else may lose some points for annoying the instructor.

Paper Presentations

- Each person will present one paper.
- There are many “For further reading” papers proposed on the reading list. They are fine choices for papers to present.
- Or you can present a paper related to your project.
- Or you can present another paper.
- Claim a paper by sending e-mail to me:
  - Tell me what paper and what date.
  - The *first* person to claim a paper gets it.
  - I’ll send you a confirmation and update the reading list.

Presentations Guidelines

- A presentation should take about 20 minutes and be like a conference talk.
- An effective conference talk is an **advertisement** for the paper:
  - State why the problem matters.
  - State the main contributions.
  - Sketch how the contributions are validated — focus on one or two *interesting* points about what they did.
  - Add your own comments, questions, and criticisms. Connect the paper with other papers that we've studied in class.
- You can prepare slides and/or use the whiteboard.
  - I *strongly* recommend giving a practice run of your talk to another student.
- Note: I'll make a few presentations like this to touch on some of the "advanced topics" (see slide 7) that we won't have time to cover in detail.

Homework

- There will be 4 ± 1 homework assignments.
- Homework problems include:
  - Trying something with real, formal verification tools.
  - Some “pencil-and-paper” problems to complement the programming.
- The first assignment should go out on Sept. 17.

Projects

The goal of the project is to produce a Master's thesis proposal including:

- A statement of what problem you are addressing.
- A review of relevant research (at least five papers)
- A simple experiment (e.g. write some code) to show that your idea will probably be feasible.
- A timeline for the thesis research and write-up.
- Identify the any resources you would need:
  - Special equipment.
  - Access to proprietry data.
  - Anything else
- Identify where the risks are in the research and how you plan to handle them.
  - If you knew the results ahead of time, it wouldn't be research.

Project ideas

- I'm very happy to see projects for any aspect of formal verification.
- I also like projects that connect formal verification with other areas.
- See

  http://www.cs.ubc.ca/~mrg/cs513/2015-1/project.html

  for some project ideas.
- I prefer individual projects, but I'll consider a proposal for a group project if:
  - A clear reason is given for why the project should be done as a group and not as separate projects.
  - Clear criteria are given for evaluating each member's contribution to the project.

Project Deadlines

- February 28: proposals due.
  - State the problem you plan to address.
  - State what approach you expect to use to address the problem.
  - List at least three papers that you plan to include in your literature survey.
- April 6: intermediate report due.
  - List the papers that you have read.
  - Describe what the progress you have made on evaluating the feasibility of your idea. For example, if you're writing a program, describe what the status of developing that code.
  - Describe any issues that have come up that could impact the project.
  - This report should be 1 or 2 pages long plus short summaries of the papers that you've read.
- April 20: Final project report due.

Project Grading

$<$ 80: You didn't do what you proposed, and you didn't try to revise the project goals if you encountered some unforeseen difficulty. Note that you can always check with me to revise the project proposal if you need to.

80 – 84: You did what you proposed, but you didn't demonstrate that you explored the topic in a way so that you learned something significant in the process.

85 – 89: You clearly learned something significant by doing the project. Make sure that your report clearly states what you learned by doing the project.

90 – 94: I learned something significant by reading your report.

95 – 100: This work is worthy of writing a paper that I expect to be a landmark in the field.

Will I survive this class?

Yes.

- Formal verification spans many areas of computer science and other fields including:
  - Mathematical logic, programming languages, digital logic design, computational complexity, computer architecture, temporal logic, robotics, differential equations, optimization.
- I expect that most of these will be new to most students in the class
  - Lectures will include many tutorials.
  - I will emphasize showing the connections between what you already know and other branches of computer science.
- The goals of the course are:
  - Give you an introduction to formal methods so you can do research in the area if you want to.
  - Give you a background so you can see how these methods are useful in other areas where you may end up doing your thesis research or working after you graduate.
  - Have fun exploring how formal methods solve challenging, real-world problems.

Binary Search: outline

- I'll do this as live code Demo.
- The proof is with ACL2.
- I'll then look at how we can simplify some parts using Z3.