"Automated Methods for Program Verification"

"Методы автоматизированной верификации программ"

Course Overview

1 The aim of the course

The aim of the course is to introduce the students to some of the most successful logic based *concepts, tools and techniques* used today in CS and IT, which are behind **a major breakthrough in the practical applications** in verification of systems and software.

These methods are proven to be of great theoretical and practical potential in CS and IT.

The topics of this course fall under the umbrella of what is called *verification*

(a) *Verification* means to verify that a system satisfies some property.

(b) The *system* can be a physical or software system.

(c) The *property* is expressed using specifications within a certain logical language.

This course is *new*. The course has been designed in accordance with the most recent trends in the program verification.
The challenge has been twofold:

(1) to select the material and design the course so that to make it meet the actual needs of the CS applications, and

(2) to do that so that to allow the students *to learn and digest* all necessary ideas to efficiently prove that the programs behave in the correct way.

This course is based on many years of the author's teaching experience in related subjects at the University of Pennsylvania, Queen Mary, University of London and University College London.

1.1 Why verification, not testing

Software engineering:

- The ultimate goal is to deliver a system doing what was intended to do, in particular free from bugs!
- Common software engineering cycle:
  specify, design, implement, debug, test,
- What is the problem with testing?

Verification is different from testing in that

(a) "Testing can only show *the presence of bugs* but never their absence" [Dijkstra], but

(b) Verification can prove *the absence of bugs* as it is based on math principles.

(c) How do we check that a system is bug free?

Answer: Static Analysis, Model Checking, Abstract Interpretation, Theorem Proving, etc.

(d) Why these techniques can do what testing cannot?

Because they are based on mathematically strong formal methods.

(e) E.g., RSA encryption is based on generating two prime numbers p and q. Could we use the polynomial

$$x^2 + x + 41$$

to generate prime numbers we need ?

We can check that for $x = 0, 1, 2, 3, \ldots, 29, 30, 31$, even 35, 36, the polynomial yield prime numbers. Can we guarantee the desired results for all positive integers x ?

1.2 Subtleties: True or False or Nobody knows

A "simple" example: $\widetilde{\mathcal{C}}_1$

```
input(x);
while (x != 1) {
  if (x is even) { x = x/2; }
  else           { x = x + 1; };
};
y = x;
output(y);
```

x	1	2	3	4	5	27	1001
y	1	1	1	·	·	·	·

Hypothesis: *For any positive integer input x, the program $\widetilde{\mathcal{C}}_1$ terminates and the resulting output y equals* 1.

TRUE!

A "simple" example: $\widetilde{\mathcal{C}}_3$

```
input(x);
while (x != 1) {
  if (x is even) { x = x/2; }
  else           { x = 3*x + 1; };
};
y = x;
output(y);
```

x	1	2	3	4	5	27	1001
y	1	1	1	·	·	·	·

Hypothesis: *For any positive integer input x, the program $\widetilde{\mathcal{C}}_3$ terminates and the resulting output y equals* 1.

Nobody knows!

2 Verification: who use it?

Huge money (and the work market !) are there - **to guarantee bug free systems**.

Software Horror Stories:

- 8 Aug 2016 - A power outage at Delta Air Lines wreaked havoc on its reservations system. the latest in a long line of computer problems
- Ariane 5 Rocket 1996. Cost $500 millions. Floating point error (overflow)
- A train stopped in the middle of nowhere (London' Docklands Light Railway) due to future station location changes after the software was deployed and reluctance to change the software.
- The Mars Climate Orbiter crashed in September 1999 because of a "silly mistake": wrong units in a program (metric mixup).
- 7 deaths of cancer patients were due to overdoses of radiation resulting from a race condition between concurrent tasks in the Therac-25 software.
- etc.

Verification: who use it?

(1) The first industries to use verification tools were hardware companies starting with Intel, Motorola to verify that there were no bugs in the logic of circuits.

(2) Most hardware companies use verification tools nowadays !

(3) Aerospace industries (airbus, NASA) use verification tools for critical components.

(4) More recently verification tools have been used in software industry, e.g. by Microsoft to eliminate bugs in device drivers (these are the main causes of bugs as they are written by third parties) !

(5) Facebook Android code is analysed by a software that implements verification ideas originating at Queen Mary, University of London and University College London.

(6) etc., etc..

3 More specifically...

The course will be delivered **in both languages**, if necessary, in order:

(a) to provide better understanding for the native-speaking people, and, at the same time,

(b) to guarantee a quicker adaptation to the international terminology, literature, the most innovative cutting edge techniques, etc., and

(c) to be prepared to understand / develop / use the most advanced automated tools in CS and IT.

In particular,

(i) We will introduce students to techniques behind most impressive verification tools, which are based on *Hoare logics* and *SAT solving*.

(ii) In its second part the course will cover *symbolic model checking* and *symbolic execution*.

3.1 Learning outcomes

The content is deliberately planned around the concept of learning outcomes or objectives. So that a student can garner some understanding of what the course should allow them to learn and that they should check (in a whatever way seems appropriate) after studying that material that they have achieved and can demonstrate those outcomes.

- Understand the aim of the course.
- Get a big picture on the materials to be covered by the course.
- Learn syntax and semantics of predicate logic with ensuring its orientation to the actual needs of computer science and information technology.
- Specify programs formally using Hoare triples
- Manually run symbolic execution rules for loop-free programs
- Use symbolic execution rules to prove loop-free programs semi-automatically
- Understand the definition of loop invariants
- Write loop invariants for simple arithmetic programs
- Prove simple arithmetic programs using loop invariants
- Understand symbolic execution for loops
- Understand an algorithm for inferring loop invariants automatically
- Apply the algorithm for simple arithmetic programs
- Understand SAT solvers
- Understand the DPLL algorithm:
- Apply Conflict-Driven Clause Learning as an efficient tool for SAT solving in practice.
- Understand how to translate programs into SAT.

4 Prerequisite, Reading Material

A minimal knowledge of logic and math is needed.

In fact, the course is *self-contained* - all basic concepts will be introduced step-by-step in class.

The pace of the course is driven not only by the syllabus but by the lecturer-students interaction.

Recommended Reading:

M.Huth and M.Ryan. Logic In Computer Science. Cambridge University Press.

Since the course is intended to follow the most recent achievements, there is no textbook which would serve the course in full extent.

Therefore, we follow the standard structure of the course: "lectures + tests", supported by a system of exercises and reading material *handed out on a weekly basis*.

The model solutions will be discussed in class and posted on the Web. Students are strongly encouraged to take part in these discussions.

On top of that, the handouts will be regularly updated to meet students' requests.

As a result, each of the students will collect the whole supporting package including handouts, practice problems, model answers to the homework assignments and midterms, etc.

5 Assessment, to be finalized in the process

(i) Mid-term tests worth 30% of the course marks.

(ii) Final exam worth 70% of the course marks.

6 Outline

(1) *Topic 1*: **Introduction to the course and basic concepts of verification**

(2) *Topic 2*: **Basic concepts of propositional logic**

Learning outcomes:
Discuss propositional connectives.

(3) *Topic 3*: **Predicate logic as a language for CS**

Learning outcomes:
Discuss several subtle examples of encoding problems into predicate logic.
Discuss the importance of distinguishing syntax and semantics.
Formally describe the syntax of predicate logic.
Formally define the domain of interpretation of predicate logic (universe and interpretation of symbols).
Informally describe the semantics of predicate logic formulas, including connectives and quantifiers.

(4) *Topic 4*: **Semantics of Predicate Logic**

Learning outcomes:
Recap on the formal syntax of predicate logic.
Bound and free variables.
Formal semantics of predicate logic.
Validity, satisifiability, etc.
Brief introduction to reasoning about predicate logic formulas.

(5) *Topic 5*: **Hoare Triples** $\{P(x, y, z)\}\ c\ \{Q(x, y, z)\}$

Learning outcomes:
Introduction to Hoare logic as the foundation of many successful techniques for formal verification of software.
Provide intuition about program correctness through examples of program specifications in Hoare logic.
Formally define the syntax and semantics of Hoare triples.
Distinguish between the notion of partial and total correctness.
The importance of auxiliary variables.
Use Hoare Triples to specify simple program in the While programming language.

(6) *Topic 6*: **Hoare Logic: Inference Rules**

Learning outcomes:
Formally define inference rules of Hoare logic for reasoning about partial program correctness.
Explain the connection between forwards and backwards axiom for assignment and weakest preconditions for assignment.
Explain how reasoning about predicate logic formulas is used for verifying programs using Hoare Logic.
Apply the Hoare Logic inference rules to prove (or disprove) correctness of loop free programs.
Use loop invariants to prove correctness of programs with loops.
Discuss the difference between inference rules for partial and total correctness.
Discuss extensions of Hoare Logic to support other programming language features, such as *pointers*.

(7) *Topic 7*: **Symbolic Execution.**

Learning outcomes:
Classic (single-path) symbolic execution.
Current applications and developments.

(8) *Topic 8*: **Hoare Logic and Symbolic Execution.**

Learning outcomes:
Recap: Syntax and semantics of while programs.
Recap: Hoare triples.
Proving Hoare triples of programs using forward symbolic execution.

(9) *Topic 9*: **Automatic Inference of Loop Invariants.**

Learning outcomes:
Understand an algorithm for generating loop invariants automatically.

(10) *Topic 10*: **SAT Solvers**

Learning outcomes:
What is SAT.
Complexity issues and simple case of Horn Clauses.
Normal forms for general case.
Tseitin transformation.
Understand the DPLL algorithm:
- unit propagation,
- pure literal elimination.

Conflict-Driven Clause Learning as an efficient tool for solving the Boolean satisfiability problem in practice.

(11) *Topic 11*: **SAT Solvers, Symbolic model checking**

Learning outcomes:
From code to propositional logic:
- single static assignment;
- loop unfolding;
- basic of software verification using SAT solvers.

(12) *Topic 12*: **Software verification using SAT solvers**

Learning outcomes:

To understand:
how to translate programs in propositional formulas;
how to translate assignments into propositional formulas;
how to translate conditional statements into propositional formulas;
the importance of using negation of assertions for verification;
the use of assumptions for verification.

7 Schedule

N	Topic	Total	In class		Self-study
			Lectures	Seminars	
1	Introduction to the course. Basic concepts of verification	10	2	2	6
2	Basic concepts of propositional logic	10	2	2	6
3	Predicate logic as a language for CS	10	2	2	6
4	Semantics of Predicate Logic	10	2	2	6
5	Hoare Triples.	10	2	2	6
6	Hoare Logic: Inference Rules	10	2	2	6
7	Symbolic Execution.	12	2	4	6
8	Hoare Logic and Symbolic Execution.	12	2	4	6
9	Automatic Inference of Loop Invariants.	14	2	4	8
10	SAT Solvers.	14	2	4	8
11	SAT Solvers, Symbolic model checking	16	2	6	8
12	Software verification using SAT solvers	16	4	4	8
	Total	144	26	38	80