Can we know that software is correct?

CMPUT 664 · Formal Verification

Jocelyn Qiaochu Chen

Before we start

Think of a piece of software you trust.

Why do you trust it?

We trust software for many reasons

- We tested it
- Other people use it
- It has not failed yet
- We inspected the code
- We trust whoever built it

Which of these is evidence of correctness?

Consider this function

```
1 def binary_search(a, target):
2     lo, hi = 0, len(a)
3     while lo < hi:
4         mid = (lo + hi) // 2
5         if a[mid] < target:
6             lo = mid + 1
7         else:
8             hi = mid
9     return lo
```

What would it mean for this function to be **correct**?

Correct with respect to what?

- Does it return an index containing `target`?
- Does it return where `target` should be inserted?
- Must the input be sorted?
- What if `target` appears more than once?
- What about an empty list?

> Correctness is meaningless without a specification.

A specification is a contract

Typically used specifications (not exhaustive) are:

Precondition

> What must be true before execution?

Postcondition

> What must be true afterward?

Invariant

> What remains true while the system evolves?

A possible contract

Given a sorted sequence a, return an index i such that

$$0 \leq i \leq |a|$$

and

$$(\forall j < i.\ a[j] < t) \wedge (\forall j \geq i.\ a[j] \geq t).$$

What is formal verification?

> The use of mathematical models and machine-checked reasoning to establish whether a system satisfies a specification.

Three ingredients:

$$\text{system} + \text{specification} + \text{reasoning method}$$

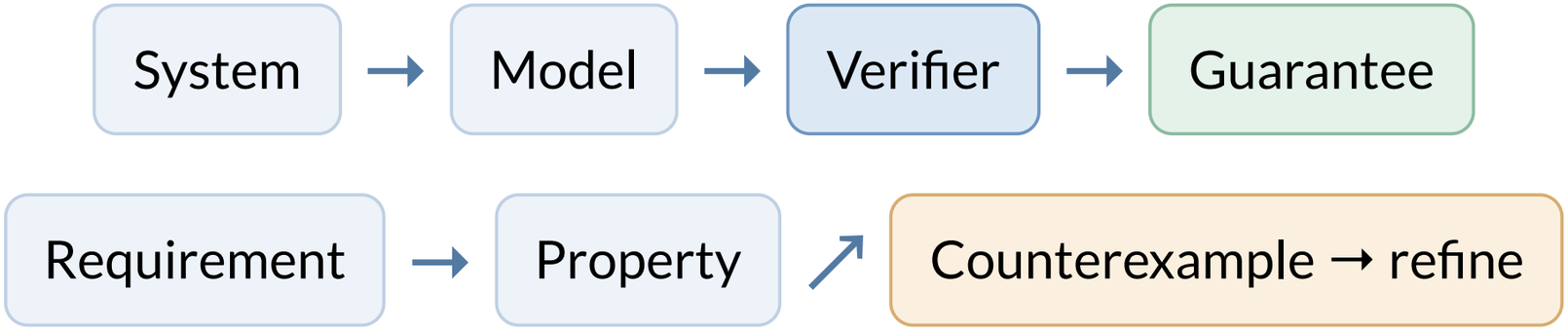
The central question

$$M \models P \ ?$$

- M: our model of the system
- P: the property we want
- $\models$: every behavior allowed by M satisfies P

How do we answer the question?

The verification loop



Verification is usually an iterative modeling process.

What can the verifier tell us?

Yes

The property follows from the model and assumptions.

No

Here is a counterexample—or a proof obligation we could not discharge.

Unknown

The method may be incomplete, too expensive, or need help.

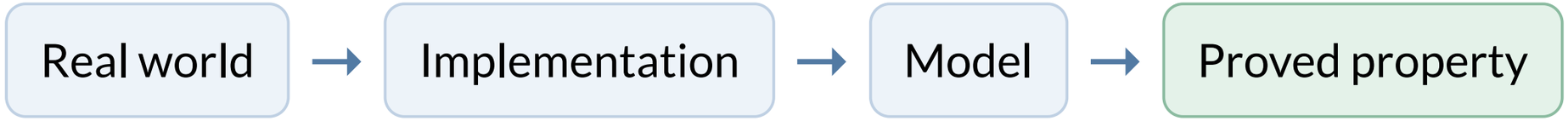
A guarantee has a boundary

> We prove properties of a model—not reality directly.

$$\text{Guarantee} = \text{property} + \text{model} + \text{assumptions}$$

A proof can be valid while the deployed system still fails.

Where can the gap be?



- The requirement can be wrong
- The model can omit behavior
- The implementation can differ from the model
- The environment can violate an assumption

Then why verify?

Verification forces us to make claims precise.

It can:

- Cover enormous or infinite state spaces
- Find subtle corner cases
- Produce machine-checkable evidence
- Preserve guarantees as systems evolve
- Expose ambiguity before deployment

Why challenging? The state-space problem

Suppose a component has 100 Boolean state variables.

$$2^{100} \approx 1.27 \times 10^{30}$$

At one billion states per second:

$$\text{more than } 10^{13} \text{ years}$$

Brute force is not a plan.

The trick: reason symbolically

Instead of trying every value of x:

$$x > 10 \land x < 5$$

ask whether **any** value can satisfy all constraints.

What answer should a solver return?

SAT and SMT solvers

SAT: Is there an assignment that makes a Boolean formula true?

SMT: Is there an assignment consistent with theories such as integers, arrays, or bit-vectors?

```
1 (declare-const x Int)
2 (assert (> x 10))
3 (assert (< x 5))
4 (check-sat)
```

Deductive program verification

Start with code and a contract.

```
1 method Abs(x: int) returns (y: int)
2   ensures y >= 0
3   ensures y == x || y == -x
4 {
5   if x < 0 { y := -x; }
6   else     { y := x;  }
7 }
```

Generate logical obligations whose validity implies correctness.

Partial vs. total correctness

Partial correctness

If the program terminates, its result satisfies the specification.

Total correctness

The program terminates **and** its result satisfies the specification.

$$\text{total correctness} = \text{termination} + \text{partial correctness}$$

Termination is a property to prove

For every input satisfying the precondition:

$$\exists y.\ \ \mathrm{Terminates}(P, x, y) \wedge \mathrm{Post}(x, y)$$

Typical evidence:

- Structural recursion
- A loop variant
- A well-founded decreasing measure

Interactive theorem proving

Define a program, prove that it terminates, and prove what it returns.

```
1 def countdown : Nat → List Nat
2   | 0     => []
3   | n + 1 => n :: countdown n
4
5 theorem countdown_length (n : Nat) :
6     (countdown n).length = n := by
7   induction n with
8   | zero => rfl
9   | succ n ih => simp [countdown, ih]
```

Lean accepts the structural recursion and checks the functional proof.

Three styles of verification

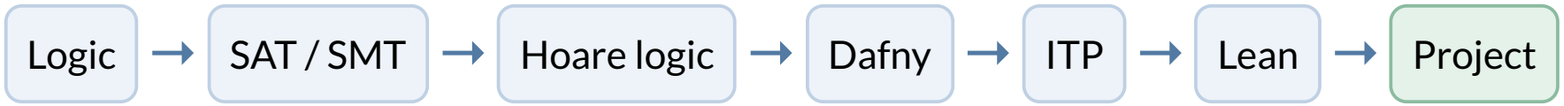
Automated reasoning	Program verification	Interactive proving
Constraints	Code + contracts	Definitions + theorems
Highly automated	Automation plus annotations	Human-guided proof
Z3	Dafny	Lean

These are complementary—not competing—approaches.

What you will (hopefully) learn to do

- Turn informal requirements into precise properties
- Model problems as logical constraints
- Read models and counterexamples critically
- Specify and verify imperative programs
- Discover and strengthen loop invariants
- Construct machine-checked proofs
- Recognize what a proof does—and does not—guarantee

The course journey



We move from automated solving toward increasingly expressive proofs—including total correctness.

What this course is not

- A survey of every formal method
- Pure mathematics detached from software
- A promise to prove arbitrary programs automatically
- A replacement for testing, review, or good engineering

> The goal is to learn when a guarantee is possible, useful, and trustworthy.

Testing and verification answer different questions

Testing	Verification
What happened on these executions?	What can happen on all modeled executions?
Runs the system	Reasons about the system
Finds concrete failures	Finds failures or establishes a property
Depends on test selection	Depends on model and specification

How you will be assessed

Component	Weight
Three programming labs	45%
Two paper reviews	10%
Final project	45%

- Labs are individual unless stated otherwise
- Five late days are shared across the labs
- No late days for paper reviews
- Late submission will not be accepted for the final project

Using AI tools

Chatbot/Coding agents are permitted.

You remain responsible for **every aspect** of your submission.

You may be asked to:

- Explain your solution
- Modify it
- Extend it

If you cannot demonstrate understanding, you may receive reduced (or no) credit.

Communication and course materials

- **Course websites:** schedules, announcements
- **Campuswire:** announcements, lecture, assignment, and project questions
- **Classroom 50:** assignment submissions
- Lectures are by-default in-person and are not recorded (There may be a few remote lectures during the semester)
- Recording or distributing course materials requires prior written consent, except under an approved accommodation

Submit your GitHub ID

Open the GitHub ID submission form

QR code for the GitHub ID submission form

Next time

Logic refresher and SAT solving

How can a machine search for truth assignments?