System “Correctness”

- A system is correct when it meets its requirements
  - “a design without requirements cannot be right or wrong, it can only be surprising”
- Verification must always start with the identification and formalization of all relevant correctness requirements

The Problem at Hand

- Finding defects in programs is hard
- This is especially true in large systems
- Concurrency only worsens the problem

If only there were a way to see into the future and determine what a program is going to do…
System Correctness

• Cannot prove a system correct in absolute sense
  – you can only prove that a system does or does not have certain specific properties
• It requires human judgment to conclude whether having or not having those properties constitutes ‘correctness’
  – getting the properties (requirements) right is as important as getting the (model of the) system right
• There is no magic wand, no blind test that could “automagically” prove an arbitrary given system ‘correct’

Correctness Requirements

• Some requirements are standard:
  – A system (e.g., an OS) should not be able to deadlock
  – No process should be able to starve another
  – No explicitly stated assertion inside a process should ever fail
• The most important requirements are application-specific:
  – System invariants, process assertions
  – Effective progress requirements
  – Proper termination
  – General causal and temporal relations on states
    • e.g., when a request is issued eventually a reply is returned
  – Fairness assumptions
    • e.g., about process scheduling

Goal of System Verification

• We are interested in determining whether design requirements could possibly be violated
• Testing is concerned with making sure that the probability of violations is minimized
• Logical correctness is concerned with possibilities, not probabilities

Impossibility >> Low Probability

Decidability

• A problem is decidable, if
  – it has a solution,
  – and the solution can be found in a finite amount of time
• Otherwise the problem is undecidable
Gödel's Incompleteness Theorem

“For any consistent formal theory that proves basic arithmetical truths, an arithmetical statement that is true but not provable in the theory can be constructed. That is, any theory capable of expressing elementary arithmetic cannot be both consistent and complete.”

Turing’s Proposition

• “There must be problems that are undecidable”

• Example: The Halting Problem (HP)
  – “Given a program and an input to the program, determine if the program will eventually stop when it is given that input.”

• Simply running the program with the given input is not a valid approach. Why?

Gödel's Theorem: “Proof”

• Consider statement
  – \( G = \) “\( G \) cannot be proven to be true”

• If \( G \) is proven to be true, there is a contradiction
  – The theory is inconsistent

• If \( G \) cannot be proven, there is at least one statement that cannot be proven
  – The theory is incomplete

HP is Undecidable

[Turing, 1936]

• Suppose that there is a solution, called \( H \)

• Program \( P \) is just a string of characters, and so \( P \) can be provided as input to \( H \)
HP is undecidable

- Construct a new program K:
  - if H outputs “loops”, then halt
  - Else loop forever
- K does the opposite of H

Model Checking

Technology to verify requirements and design for a variety of real-time embedded and safety-critical systems

The Basic Idea

- Model Checking is an automated and exhaustive correctness verification technique for finite state concurrent systems
  - User does not need to be a formal methods expert
  - Unlike testing, model checking can show absence of errors
  - If the property cannot be verified then a counter-example path is provided
Model Checking

- Model (system requirements)
- Model Checker
- Specification (system property)
- Answer
  - Yes, if model satisfies Spec
  - Counterexample, if not

How a Model Checker Works

- Enumerate all possible execution traces that a system could take
- Evaluate the specification in each of the traces to see if it holds

Characteristics

- Characteristics of system models which favor model checking over other verification techniques
  - Ongoing input/output behavior (not: single input, single result)
  - Concurrency (not: single control flow)
  - Control intensive (not: lots of data manipulation)

Example Systems

- Control logic of hardware designs
- Communication protocols
- Device drivers
Describing System Model

While the choice of system model is important for ease of modeling in a given situation, the only thing that is important for model checking is that the system model can be translated into some form of state-transition graph.

State Explosion

Translation from system description to a state Diagram usually involves exponential blow-up

Example: $n$ Boolean variables $\Rightarrow 2^n$ states!

Solution: Use Kripke structures

State-Transition Graph

System Properties

Safety

$Nothing bad happens$

Liveness

$Something good happens eventually (but we don’t know when)$
### Safety and Liveness

**Safety:** those properties whose violation *always* has a finite witness

("if something bad happens on an infinite run, then it happens already on some finite prefix")

**Liveness:** those properties whose violation *never* has a finite witness

("no matter what happens along a finite run, something good could still happen later")

### Temporal Logic

- A formalism for describing ordering of states/events
- Input/output pairs are good for transformational systems
- Temporal logic is good for reactive systems, which continuously interact with the environment

### Expressing Properties

- **Temporal Logic**
- **Basic Operators**
  - □ - Always (G)
  - ◇ - Eventually (F)
  - ∃ - Exists (E)
  - ∀ - Forall (A)
  - ○ - Next (X)

### Temporal Logic

- Example properties for an ATM
  - Always asks for the password
  - If withdrawal is selected and there is enough money then eventually the requested amount is delivered
  - Deposit is not complete until the envelope is inserted
Temporal Logics

- Linear Temporal Logic (LTL)
  - The property must be satisfied by each path of the transition system
- Computation Tree Logic (CTL)
  - System computation is perceived as a tree and the computation tree should obey the branching dictated by the property

- The two are not comparable
  - There are CTL formulas that are inexpressible in LTL and vice versa

Explicit-State Model Checking

Program or High-level Design

Correctness Property

Model Checker

Is the bad state reachable in $A_1 \times \neg A_2$?

Not correct! And here is why:

State1 (initial)
State2
State3
...
Bad state

Correct!

Yes!

No!

Symbolic Model Checking

Program or High-level Design

Correctness Property

Model Checker

(P: temporal logic formula)

Are all initial states in $S_0$, the set of states that satisfy $p$?

Yes!

Not correct!

No!

Not correct!

Some Tools

- Explicit-state model checkers
  - Spin (Bell Labs)
  - Java Pathfinder (NASA Ames)
  - Verisoft (Bell Labs)
- Symbolic model checkers
  - SMV, nuSMV (CMU)
  - ALV (UC Santa Barbara)