Alcino Cunha

SPECIFICATION AND MODELING

INTRODUCTION

Universidade do Minho & INESC TEC

2019/20

MOTIVATION

THIS COURSE IN A NUTSHELL

- Languages and tools for (formal) software design:
  - Languages to *model* the system being designed
  - Languages to *specify* the expected properties
  - Techniques and tools to *analyse* the design

FORMAL SOFTWARE DESIGN

- The design is a high-level abstraction of the desired system
- A programming language is not adequate for software design
- The language of mathematics, logic, is a much better alternative
- It enables a formal approach to software design

Leslie Lamport

“If you’re not writing a program, don’t use a programming language”

TYPICAL ANALYSES

- Simulate the design to validate and elicit requirements
- Check consistency of requirements
- Verify expected properties

TARGET APPLICATIONS

- ~~Sequential algorithms~~
- Reactive systems
- Distributed protocols

SEQUENTIAL ALGORITHMS

- Specification with pre- and post-conditions
- Deductive verification with Hoare logic

dafny Microsoft Research

Is this program correct?

```
method Mul(x: int, y: int) returns (r: int)
  requires 0 <= x && 0 <= y
  ensures r == x*y
{
  r := 0;
  var i := 0;
  while (i < y)
    invariant r == x*i
    invariant i <= y
  {
    r := r + x;
    i := i + 1;
  }
}
```

REACTIVE SYSTEMS

- Typically non terminating systems reacting to environment
- Cannot be specified with pre- and post-conditions
- Non-determinism due to environment action
- Specifications can be very complex temporal properties
- Not amenable for deductive verification

DISTRIBUTED PROTOCOLS

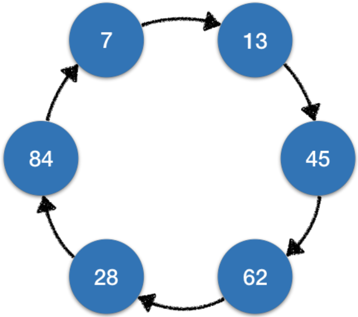
- Several processes running concurrently in independent processors
- Communicating with message passing
- Local computation with simple algorithms
- Non-determinism due to interleaving
- Specifications are mostly invariants and simple progress properties
- Deductive verification possible but not easy
  - Most invariants are non inductive
  - Must specify variants to verify progress

MODEL CHECKING

- Fully automatic verification technique for temporal properties
- Either the specification is true or a counter-example is returned
- No need to guess inductive invariants or variants
- But unlike deductive verification it requires a finite state space

EXAMPLES

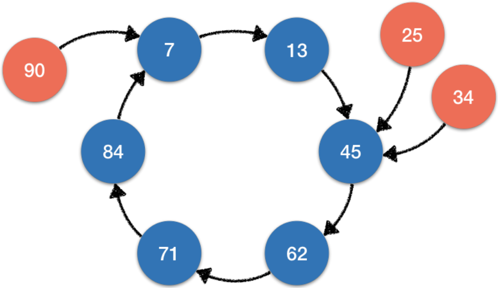
LEADER ELECTION IN A RING



Verify the correctness of the protocol:

- One leader will be elected

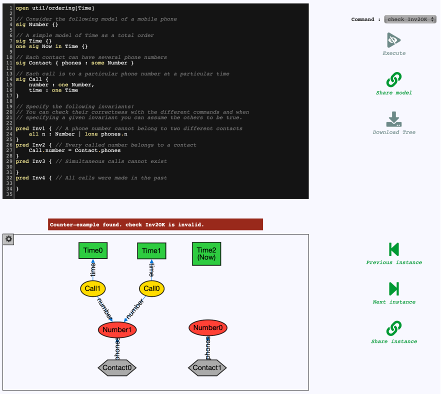
CHORD DISTRIBUTED HASH-TABLE



Explore variants of the protocol and verify correctness:

- If joins and failures cease, the network will eventually become a ring

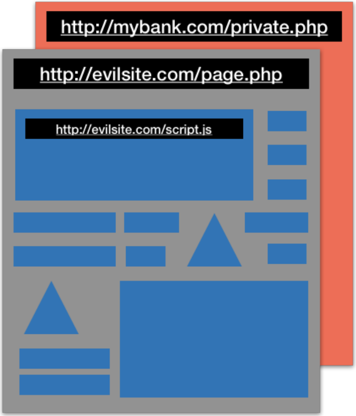
ALLOY4FUN



Explore design alternatives and elicit data invariants:

- Non-shared stored models can have at most one derivation

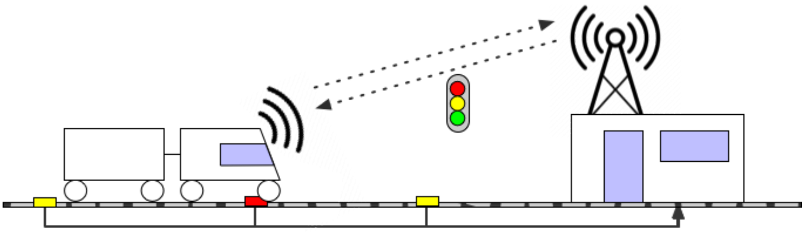
SAME ORIGIN POLICY



Understand and verify the policy:

- Resources can only access resources from the same origin

HYBRID ERTMS/ETCS LEVEL 3



Verify the design of a railway traffic management system:

- Assigned movement authorities are safe

SYLLABUS AND ASSESSMENT

LOGICS

First-order logic

The fundamental logic to specify properties about states

Relational logic

A variant of FOL better suited for software design, where the state is typically described by *relationships* between concepts or objects

Temporal logic

A logic to specify properties about behaviours

ANALYSIS TECHNIQUES

Model-checking

Automatic verification of temporal properties

Model-finding

Automatic generation of structures satisfying a set of constraints

MAIN LANGUAGES AND TOOLS

Alloy

Native support for *sets* and *relations*, relational logic, and model-finding
Good for the design of complex (graph-like) structures

Electrum (soon Alloy 6)

Extends Alloy with temporal logic and model-checking
Good for the design of systems with complex structures and many configurations

OTHER LANGUAGES AND TOOLS

SMV

The quintessential model-checker, with support to various temporal logics
Good for the design of simple reactive systems or as a back-end analysis tool

TLA+

Supports many data-types and (limited) temporal logic specifications
Good for the design of distributed and concurrent algorithms

ASSESSMENT

- One individual test (60%, >= 8)
- Several in-class individual assignments (20%)
- One take-home group assignment (20%)
- The exam replaces the test only