Distributed Systems Programming F29NM1

Formal Verification

Andrew Ireland

School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Overview

- Focus on the process of formal verification.
- Survey the various representations, techniques and tools that support formal verification.

The Process of Formal Verification

- The following phases can be seen as defining the process of formal verification:
  - Requirements capture;
  - Modelling;
  - Specification;
  - Analysis;
  - Documentation.
- Note that in reality some of these phases may overlap, in addition the overall process should be seen as iterative rather than sequential.
- We will focus on the middle three phases ...

Modelling Phase

- Mathematical models can be developed for both discrete and continuous domains — we will focus upon discrete models.
- Mathematical models for discrete domains:
  - Functional & relational models;
  - Abstract state machine models;
  - Automata-based models;
  - Object-oriented models.
- What typically distinguishes models within formal verification from other mathematical models is the notion of an underlying computational model, *e.g.* function composition, interleaving model, ...

Functional & Relational Models

- Functional: Algorithms modelled as recursive functions, *e.g.*

  $$Device(a, b, c) \equiv fun_2(fun_1(a, b), c)$$

  - Reasoning relatively easy;
  - Function composition supports rapid prototyping.
- Relational: Algorithms modelled as relations, *e.g.*

  $$Device(a, b, c, d) \equiv \exists p.\ rel_1(a, b, p) \wedge rel_2(p, c, d)$$

  where $\exists$ denotes "there exists" and $\wedge$ denotes "logical and".
  - More natural than functional representation, *e.g.* variables correspond to "wires" within the model of a circuit;
  - More expressive than functional representation, *e.g.* feedback loops can be modelled;
  - Reasoning and rapid prototyping are harder.

Abstract State Machine Models

- A state machine model contains:
  - An abstract representation of **system state**, *e.g.* values associated with the `program counter` and `registers`;
  - A set of state operators which define a transition relation between the current and next states.
- For a given input-state pair the state transition operators determine the associated output and next state, *i.e.*

  $$Model : (Input \times State) \rightarrow (Output \times State)$$

  Supports rapid prototyping but reasoning can be cumbersome if the state becomes too large, *e.g.* many intermediate system properties may need to invented.

Automata-based Models

- An automata-based model is a finite-state transition system:
  - set of states;
  - set of state-to-state transitions based upon input stimuli.

  Automata may be **deterministic** or **non-deterministic**,
- *-Automata: the conventional notion of automata where there exists explicit initial and final states, *i.e.* recognizes finite sequence of stimuli. Acceptance corresponds to final state.
- ω-Automata: an automata which contains an explicit initial state but no final state *i.e.* recognizes an infinite sequence of stimuli (reactive systems). Acceptance requires a different criteria, *i.e.* **language inclusion**; **state exploration**, **reachability analysis** (more later).

Object-oriented Models

- In an object-oriented model a system is represented by a structured collection of classes and objects which support **encapsulation** and **inheritance**.
- Many informal object-oriented methodologies, the emerging "market leader" is the Unified Modelling Language (UML).
- The roots of object-oriented modelling are in simulation rather than formal mathematics so there tends to be a lack of precision when it comes to semantics and formal analysis. However, there have been attempts at combining object-oriented features within existing formal frameworks, *e.g.* LOTOS and VDM (to give VDM++).

Developing Models

- Two important concerns when developing mathematical models:
  - Abstraction: crucial for mastering complexity, but care must be taken to ensure that no key characteristic of the "real" system is lost, *e.g.* modelling a railway system without the possibility of a driver passing a signal at danger has obvious limitations in terms of safety predictions.
  - Expressiveness vs analytical power: the more expressive a representation one uses to model a system the harder the process of analysis becomes, *e.g.* relations are more expressive than functions but formal reasoning becomes harder.

Specification Phase

- Complete verification involves two models of a system where one denotes the specification (abstract) and the other denotes the implementation (concrete). The intention being that both models are in some formal sense equivalent. Risk of errors being propagating from abstract model to concrete model.
- Property verification involves a single model of the system and a set of desired properties that are to be established with respect to the system model. Risk of under-specification, however, simple abstract properties can be very effective at identify design flaws.
- Hybrid is possible and desirable, *i.e.* develop key properties for both the "specification" and "implementation" models.

Logics for Specification

- Classical propositional logic, *e.g.*

  $((P \rightarrow Q) \wedge P) \rightarrow Q$

- Classical first-order predicate calculus, *e.g.*

  $\forall X.\exists Y.(number(X) \wedge number(Y)) \rightarrow greater(Y, X)$

  Also higher-order versions which support quantification over functions and predicates.

- Temporal reasoning:

  $\Box(P \rightarrow \Diamond Q)$

  This assert that "henceforth whenever P is true then eventually Q will become true".

  Many more logics ...

Specification of Correctness Properties

- Partial correctness:

  $(Pre(x) \wedge terminates(Prog(x, y))) \rightarrow Post(x, y)$

- Total correctness:

  $Pre(x) \rightarrow (terminates(Prog(x, y)) \wedge Post(x, y))$

- Safety properties:

  $\Box P$

  property P must **always** hold (system invariant), *e.g.* **mutual exclusion** and **absence of deadlocks**.

- Liveness properties:

  $\Diamond P$

  property P must **eventually** hold, *e.g.* **absence of resource starvation**.

Formal Analysis

- Direct execution & simulation.
- Equivalence checking (finite-state).
- Model checking (finite-state).
- Theorem proving (infinite-state).

Direct Execution & Simulation

- Allows a designer to **observe** the behaviour of a system model with respect to particular input stimuli.
- Direct execution and simulation are not formal analysis techniques, they complement formal analysis.
- However, within the context of formal methods, direct execution and simulation are possible if the modelling language is executable, *e.g.*
  - The modelling language associated with the Nqthm theorem prover is a subset of common LISP;
  - The specification language VDM has an executable subset which has been exploited commercially (see VDM-SL Toolbox produced by IFAD).

Equivalence Checking

- An **equivalence checker** determines whether or not two finite state representations perform the same function.
- In terms of formal verification, if a **reference design** and a candidate **implementation** can be represented by finite states then equivalence checking is applicable.
- Equivalence checking works well in hardware design during the final tuning and optimization phases.
- Equivalence checking assumes that the reference design is bug free!

Model Checking

- **Model checking** is more powerful than equivalence checking. An equivalence checker compares finite states while a model checker allows us to verify temporal properties of a finite state representation.
- The basic idea is that a finite state model of a system is systematically explored in order to determine whether or not a given temporal property holds.
- The technique is fully automatic and delivers a counter example if the specified property does not hold.
- Because a complete state space is generated for a given model, the analysis may fail due lack of memory if the model is too large. This **state explosion problem** can be tackled via abstraction.

Theorem Proving

- **Theorem proving** is more powerful than model checking. Theorem proving allows the verification infinite models, *i.e.* parameterized designs such as a N-bit carry adder or a bus architecture with N-processors.
- The basic idea is that both the model and specification (refined model or property) are represented within a suitable logic. Theorem proving is the process of showing that the model satisfies the specification by means of formal proof, *i.e.* a sequence of statements each of which is an instance of an **axiom** or follows from earlier statements by means of a **proof rules**.
- Avoids the state explosion problem but incurs a combinatorial explosion with respect to the applicability of axioms and proof rules, at best a semi-automatic technique.

Towards Integration of Analysis Techniques

- All the formal analysis techniques have strengths and weaknesses.
- Industrial strength applications typically require a variety of analysis techniques.
- Current trend is towards integration, *e.g.*
  - The Stanford TEmporal Prover (STEP) combines both model checking and theorem proving capabilities;
  - A model checker has been integrated within the PVS theorem prover (SRI).
- Ultimately formal analysis will need to be embedded within conventional computer-aided design frameworks ...

SPIN: Design Verification System

- Modelling, specification and analysis of distributed systems.
- Promela – PROcess MEta LAnguage:
  - Influenced by Dijkstra's guarded command language, Hoare's process algebra CSP and has C-like syntax;
  - Process communication modelled via message channels (asynchronous and synchronous).
- SPIN – Formal analysis tool for Promela programs:
  - Supports simulation, either random or interactive;
  - Supports formal verification, *i.e.* deadlock freedon; unexecutable code; non-progress execution cycles; model checking for linear time temporal properties.

Summary

Learning outcomes:

- Gain an understanding of the various representations, techniques and tools associated with the formal modelling, specification and analysis of system designs.
- Overview of the SPIN design verification system.

Recommended reading:

- `http://www.comlab.ox.ac.uk/archive/`
- Formal Methods Specification and Analysis Guidebook for the Verification of Software Systems: Volume II A Practitioner's Companion, Published by NASA, see `http//eis.jpl.nasa.gov/quality/Formal_Methods/`

Note: All web links available via module homepage:

`http://www.macs.hw.ac.uk/~air/spin/`