# Automated Reasoning for Software Engineering

**L. Georgieva and A. Ireland**

School of Mathematical and Computer Sciences

Heriot-Watt University

Lilia: Office G.50

Email: lilia@macs.hw.ac.uk

# Lecture 1: Overview

- To explore topics within automated reasoning as applied to software engineering.
  - interactive proof tools (PVS);
  - model checking;
  - temporal verification of distributed communicating systems;
  - simulation versus verification;
  - model checking applications within automated software engineering;
  - the complementary roles of theorem proving and model checking.
- The module is divided into two themes:
  - theorem proving
  - model checking.

# Module Overview

**Lecturers:** Andrew Ireland (G.57) & Lilia Georgieva (G.50)

`air@macs.hw.ac.uk` `lilia@macs.hw.ac.uk`

**Themes:** Theorem proving weeks 2, 3, 6, and 7; Model checking weeks 4, 5, 8, and 9 (revision week 10)

**Lectures:** Tue-10.15 in 3.06; Thu-12.15 in 3.07; Fri-09.15 in G.44

**Labs:** Thu-12.15 in 2.50 (Linux Lab)

**Coursework:** Two assignments, one for each part of the module (20%):

Assignment 1: Out: Week 3, Due: Week 7;

Assignment 2: Out: Week 5, Due: Week 9.

**Examination:** After Easter - questions from both parts (80%)

**Materials:** Teaching materials are on the web:

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

`http://www.macs.hw.ac.uk/~lilia/ar/`

# History

- Prehistory: transformational programs and theorem proving.
- Early 80's: foundations.
- Late 80's: first tools.
- Early 90's: state space explosion.
- Late 90's: the boom.
- Now: how can model checking be applied to software?

In this module we will discuss challenges and approaches of applying model checking and theorem proving to software.

# Verification

- Verification framework: specification, design, implementation, verification.
- Verification: we aim to check whether all possible behaviors of a system are compatible with the specification.
- Testing can find errors, verification can prove their absence.

# Prehistory

- Early computer programs were designed to compute something (accounting, scientific computing).
- Transformation form initial to final state.
- Specification: precondition and postcondition.
- Formal verification: paper and pencil, first theorem provers (CAV)

# Theorem proving

- Goal: to automate logical reasoning.
- Verification using theorem proving
  - The implementation is represented by a logical formula I (Hoare's logic).
  - The specification is represented by a logical formula S.
  - Question: Does I imply S hold?
  - Syntactic level proof.
- General approach: applicable to many programs and properties.
- However: most proofs are not fully automatic.

# Transformational versus reactive programs

- Transformational program computes something.
- Reactive program:
  - controls something.
  - continually interacts with its environment.
  - FSM, state space, behavior described in terms of sequences of states.
  - language for temporal properties: temporal logic.

# Linear-time Temporal Logic (LTL)

- Specify properties of infinite sequences of states (or transitions).
- Temporal operators include: G (always), F (eventually) and X (next).
  Example: $\mathsf{G}(\mathsf{p} \longrightarrow \mathsf{Fq})$
- "Does $M$ satisfy $\varphi$ ?" = model checking
  - For $\varphi$ in LTL, do all infinite computations of $M$ satisfy $\varphi$?
  - For $\varphi$ in BTL, does the computation tree of $M$ satisfy $\varphi$?
- Algorithmic issues: efficient decision procedures exist?
  - Proof can be carried out at semantic level, via state-space exploration.
  - for BTL, SAT is EXPTIME-complete, but model checking is linear!

# Model checking framework

- Components:
  - implementation (program) = an FSM.
  - specification (property) = a temporal logic formula.
  - comparison criteria = defined by semantics of the temporal logic.
  - algorithm = evaluates the formula against the FSM.
- Model-Checking Research in the 80's:
  - various temporal logics: linear-time, branching-time.
  - relationship between temporal logics and classes of automata (LTL and word automata; BTL and tree automata?)
  - classes of temporal properties (safety, liveness)
  - model checking is automatic but (essentially) restricted to finite-state systems.
  - many reactive systems can be modeled by FSMs!

# Tools

- Examples: CAESAR, COSPAN, CWB, MURPHI, SPIN.
- Differ by specification language, implementation language, comparison criterion, and/or verification algorithms,but all based on systematic state-space exploration.
- Using a temporal logic is not mandatory.
- Many "model-checking" tools do not support a full temporal logic.
- From now on, no distinction here between model checking and systematic state-space exploration.
- Logic is a powerful theoretical tool (characterizes classes of properties).
- Logic can be very useful in practice too (concise and expressive).
- First success stories in analyzing circuit designs, communication protocols, distributed algorithms!

# Model checking in practice

Model checking can be very useful!

- Main strength: model checking can detect subtle design errors.
- In practice, formal verification is actually testing because of approximations:
  - when modeling the system,
  - when modeling the environment,
  - when specifying properties,
  - when performing the verification.
- Therefore "bug hunting" is really the name of the game!
- Main goal: find errors that would be hard to find otherwise.

# Limitations of model checking

- FSM (=state space) can itself be the product of smaller FSMs.
- Model checking is usually linear in the size of the state space, but the size of the state space is usually exponential (or worse) in the system description (program).
- State-space exploration is fundamentally hard (NP, PSPACE or worse).
- Engineering challenge: how to make model checking scalable?

# Divide-and-conquer approaches

- Abstraction: hide/approximate details.
- Compositionality: check first local properties of individual components, then combine these to prove correctness of the whole system. Algorithmic approaches:
- "Symbolic verification": represent state space differently (BDD).
- State-space pruning techniques: avoid exploring parts of the state space (partial-order methods, symmetry methods).
- Techniques to tackle the effects of state explosion (bit-state hashing, state-space compression, caching).
- Result: Several order of magnitudes gained!

# Hardware verification

- Hardware verification is an important application of model checking and related techniques.
- The finite-state assumption is not unrealistic for hardware.
- The cost of errors can be enormous (Pentium bug).
- The complexity of designs is increasing very rapidly (system on a chip).
- However, model checking still does not scale very well.
- Many designs and implementations are too big and complex.
- Hardware description languages (Verilog, VHDL) are very expressive.
- Using model checking properly requires experienced staff.

# Model checking and software

- Analysis of software models: (e.g., SPIN)
- Analysis of communication protocols, distributed algorithms.
- Models specified in extended FSM notation.
- Restricted to design.
- Analysis of software models that can be compiled: (SDL, VFSM)
- Same as above except that FSM can be compiled to generate the core of the implementation.
- More popular with software developers since reuse of "model" is possible. Analysis still restricted to "FSM part" of the implementation.

Challenge: how to apply model checking to analyze software in programming languages (C, C++, Java) and real size of code (100,000's lines)?

# Theorem proving for software engineering

- Theorem proving (automated deduction) is:
  - logical deduction performed by a machine.
  - at the intersection of three areas:
    * mathematics: motivation and techniques;
    * logic: framework and reasoning techniques;
    * computer science: automation techniques;
  - extensively studied.

# Theorem proving: motivation

- Depth
  - depth: the problem requires mathematical insight (pure mathematics, Robinson's conjecture, Fermat's theorem)
  - complexity: shallow problem, many cases, usually in computer science.

# Theorem proving: applications

- Formalizing mathematics;
- Discovery of proofs of mathematical conjectures:
  - provers for geometry
  - computer algebra systems
- Software and hardware productivity and reliability systems
  - verification of prototypes
  - implementations
  - automatic program synthesis from specifications;
- Formalizing semantics of programming languages:
  - properties of the semantics;
  - verification of interpreters and compilers;
  - self validating compilers (ongoing research).

# Prototype verification systems

- Based on a rich specification language ( higher-order logic + dependent types + inductive types)
- One "programs" a prototype of the implementation
- Skill required to find a good abstraction
- Then one can "test" the prototype: E.g., the prototype is "well-typed" E.g., prove that it satisfies certain desired requirements
- This is a way to learn about the problem to be solved
- Use a proof assistant for this purpose

# Using proof assistant

- The human does the hard work
  - Formulate lemmas
  - Select the induction principle
  - Guide case splitting
- The proof assistant does the bookkeeping
  - Make sure we do not overlook cases
  - Make sure the proof rules suggested are applicable
  - Record and pretty-print the proof

# Using proof assistant

- Typical interaction:
  - Proof assistant shows the current assumptions + goals
  - User instructs the assistant to focus on a goal
  - User decides what is the next step
  - Rewrite an assumption using a forward proof rule
  - Rewrite the goal using a backward proof rule
  - This either proves the goal or produces a new subgoal
  - Iterate until no more subgoals
- Often the user has to remember complicated rule names
- Grind in Prototype Verification System (PVS) discharges many small subproofs
- Many assistants are programmable and partially automated.
- Examples: PVS, HOL, Lego, Touchstone;

# Automatic synthesis of code

- Specifications: requires Pre(x) ensures Post (x, x')
- Specification is implementable.
- Prove this fact with a theorem prover.
- "run" the proof: given x, construct x'.
- The algorithm is extracted from the proof strategy.
  - lemmas → auxiliary functions.
  - case split → conditional.
  - induction → (primitive) recursion.
- This is done frequently in Coq.
- Must have a complete specification.
- Running proofs might not be efficient.

# Soundnes and completeness

- Soundness: If the theorem is valid then the program meets specification.
- Completeness: If the theorem is provable then it is valid.

# From theorems to proofs

- Proving theorems is hard.
- Use an interactive theorem prover.
- Human must put the annotations and drive the prover.
- Or, use an automatic theorem prover.
- There is still interaction for refining the annotations.
- Automatic provers use heuristics. Hard to predict the outcome, unintuitive.
- But there are special cases in which automated theorem proving is very effective.

# Theorem proving: conclusions

- Theorem proving strengths
  - very expressive
- Theorem proving weaknesses
  - too ambitious: sacrifice soundness.
  - too hard to use/understand: bring it closer to typing.
  - a great toolbox for software checking.
  - symbolic evaluation.
  - satisfiability procedures.

## We will study:

- Interactive proof tools (proof assistants):
  - offer to prove theorems step by step;
  - user has to select an appropriate command;
  - each step that the prover offers is logically sound;
  - granularity varies;
- Theory behind higher order theorem provers:
  - deductive calculi
  - data types;
  - typed lambda calculus versus higher order logic;
- Applications of PVS to small functional programs.