# 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) (!);
  - automated proof tools (SPASS) (!)
  - simulation versus verification;
  - model checking applications within automated software engineering;
  - the complementary roles of theorem proving and model checking. (!)
  - proof systems (!)
- 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 3, 4, 7, and 8; (revision week 10)

**Lectures:** Tue-11.15 in 3.07; Fri-9.15 in G.44;

**Labs:** Fri-15.15 in 2.50 (Linux Lab)

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

Assignment (theorem proving): Out: Week 7, Due: Week 10.

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

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

`\verb+http://www.macs.hw.ac.uk/~lilia/ar/ar.html`

# 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 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

- 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.

# 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.

# Soundness 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.

# Interesting problems

- Halting problem: it is impossible to write a diagnostic program that will tell you if a given program will terminate.
- Gödel's incompleteness theorems: any formal system that includes arithmetic is either
  - incomplete (there are some properties that are true but cannot be proved) **OR**
  - inconsistent (contains one or more contradictions that allow you to prove properties that are false).

Safety critical systems: failure results in physical injury, loss of life, financial loss.

Application areas: aerospace, medical equipment, process control.

# Example: reactor shutdown system (SDS)

SDS is a watchdog system that

- monitors system parameters.
- shuts down if it observes bad behaviour.

Example: if parameters exceed certain set points: shut down the reactor.

# Safety considerations

- Check for short circuits or sensor failures.
- Use dead-band to eliminate “chatter”.
- Increase the operating margin by power dependant set points.
- Identify unreliable operating regions.
- Use multiple sensors to improve reliability.

# The process

- Multiple reviewers do:
  - software requirements specifications review;
  - software design description review;
  - code review.
- Testing:
  - unit testing: each individual program separately.
  - software integration testing: components when they are combined.
  - validation testing: test the system against the requirements.

Logic: unambiguous, precise language for specification.

# Is it enough?

- Incorrect design despite multiple reviewers.
- Testing cannot cover all possible cases.
- Minor changes result in another extensive and expensive round of testing.

Solution: prove that the design implements the specification, e.g.

- Theorem proving: use PVS to prove that

$$\text{for all inputs } x \;:\; \mathit{Spec}(x) = \mathit{Design}(x)$$

- Model checking: verify automatically that Design is a model of Spec written as a logical formula.

Advantages: independent system check, not affected by the expectations of the reviewer; domain coverage, automation.

# Example

Example:

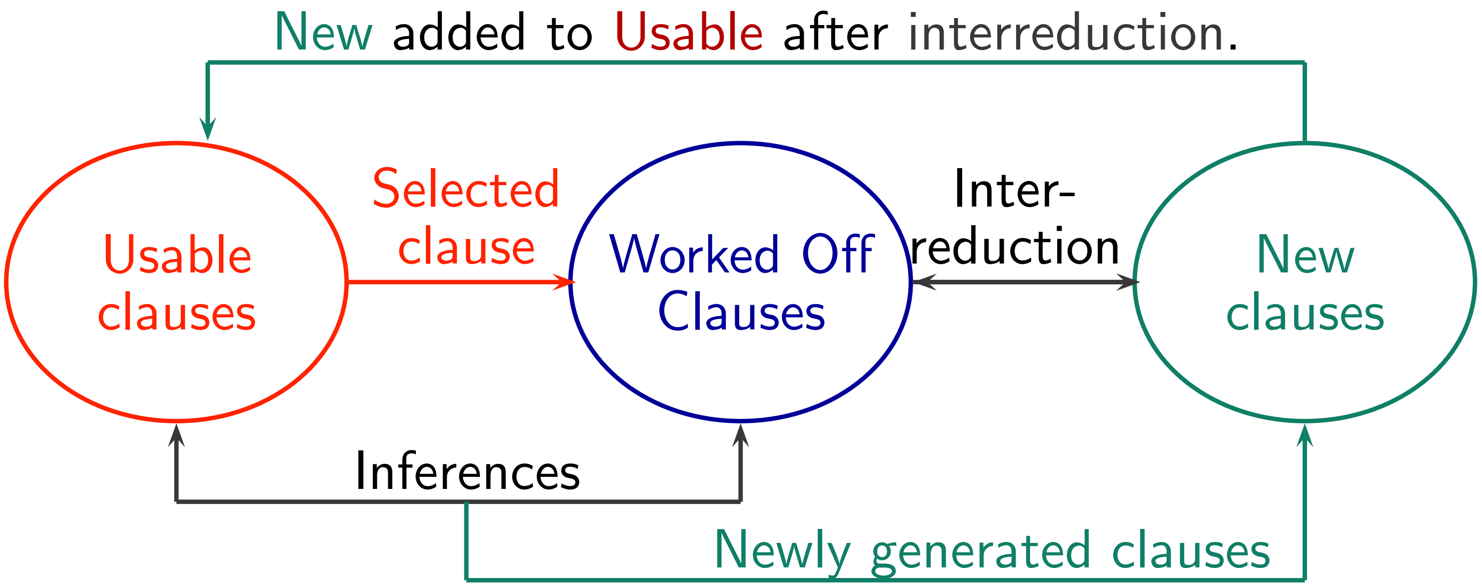
- the Pentium$^{TM}$ bug could have been detected by computer aided verification tools.
- CAV was used to prove the correctness of the suggested fix.
- PVS has been used in similar cases.

# Basic inference loop of a automated theorem prover

Implemented in: Gandalf, SPASS, OTTER, Vampire.

Input: clausal set.

Output on termination: a proof of unsatisfiability or a saturated clause set.



# 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;
- Automated proof tools.
- Theory behind higher order theorem provers:
  - deductive calculi
  - data types;
- Applications of PVS and SPASS.