# 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: Continued

# 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.

# 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 $\mathrm{Pentium}^{TM}$ 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.

# 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

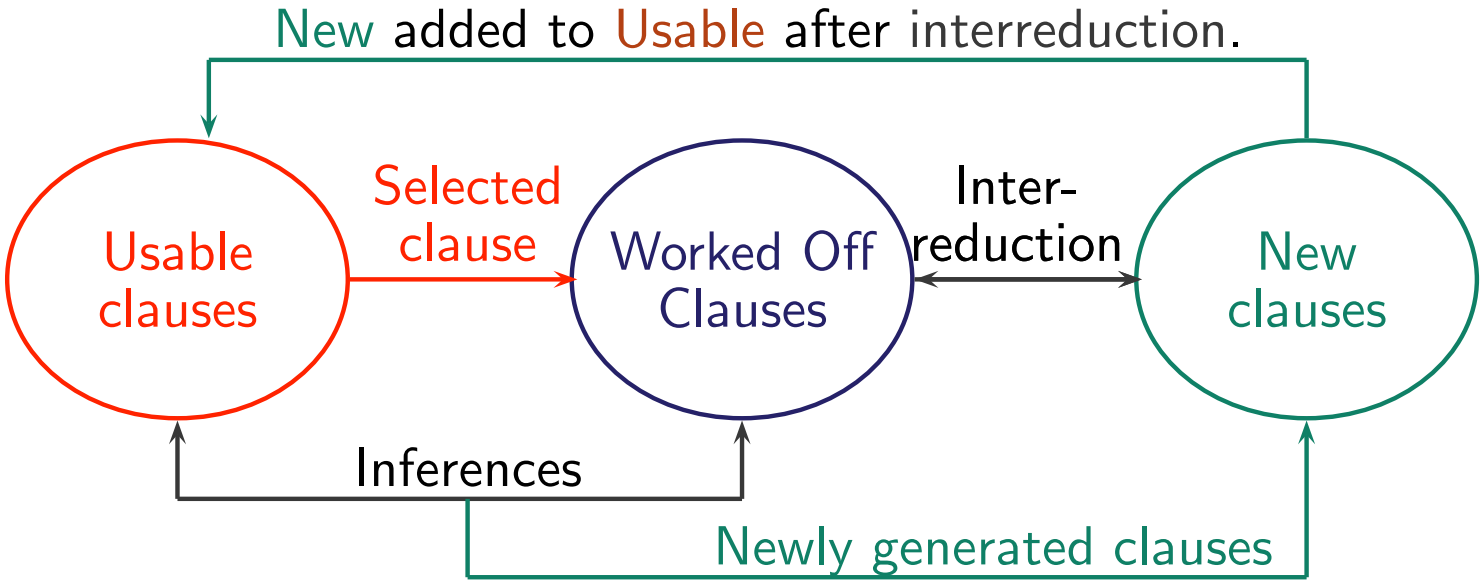
- Formalizing mathematics;
- Discovery of proofs of mathematical conjectures:
  - provers for geometry
  - computer algebra systems
- Software and hardware productivity and reliability systems
  - verification of prototypes
  - implementations
  - automatic program synthesis from specifications;
- Formalizing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

# Basic inference loop of a saturation theorem prover

Implemented in: Gandalf, SPASS, OTTER, Vampire.

Input: clausal set.

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



# 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: <u>requires</u> Pre(x) <u>ensures</u> 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 $\longrightarrow$ auxiliary functions.
  - case split $\longrightarrow$ conditional.
  - induction $\longrightarrow$ (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.

# 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.