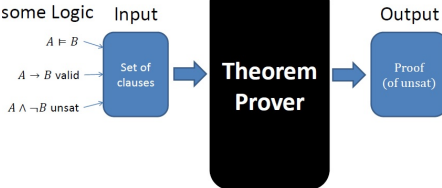


# Theorem Proving

## Automated Theorem Proving

Reasoning  
question in  
some Logic



Lucien Ngalamou

# Outline

A Framework for Software Verification

Introduction to Theorem Proving (Automatic Theorem Proving)

Hoare Logic

Class Activity

The KeY Project

# A Framework for Software Verification

- ▶ Convert the informal description  $R$  of requirements for an application domain into an "equivalent" formula  $\phi_R$  of some logic;
- ▶ Write a Program which is mean to realize  $\phi_R$  in the programming environment supplied by your company, or wanted by the particular customer;
- ▶ Prove that the program  $P$  satisfy the formula  $\phi_R$ .

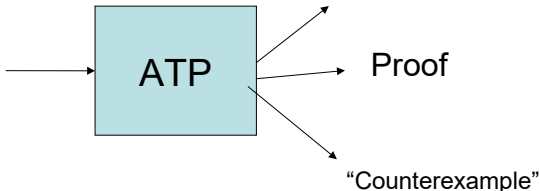
# What is an automated theorem prover?

---

Input

Output

Theorem



## Example theorems

---

- Pythagoras theorem: Given a right triangle with sides A B and C, where C is the hypotenuse, then  $C^2 = A^2 + B^2$
- Fundamental theorem of arithmetic: Any whole number bigger than 1 can be represented in exactly one way as a product of primes

# The model checking approach

---

- Create a model of the program in a decidable formalism
- Verify the model algorithmically
- Difficulties
  - Model creation is burden on programmer
  - The model might be incorrect.
- If verification fails, is the problem in the model or the program?

# The axiomatic approach

---

- Add auxiliary specifications to the program to decompose the verification task into a set of local verification tasks
- Verify each local verification problem
- Difficulties
  - Auxiliary spec is burden on programmer
  - Auxiliary spec might be incorrect.
- If verification fails, is the problem with the auxiliary specification or the program?

## Example Theorem

---

- The program “ $z = x; z = z + y;$ ” computes the sum of ‘ $x$ ’ and ‘ $y$ ’ in ‘ $z$ ’ according to the semantics of  $C$
- Program-Semantics  $\Rightarrow$  Specification



# Theorem

---

- Theorem must be stated in formal logic
  - self-contained
  - no hidden assumptions
- Many different kinds of logics (propositional logic, first order logic, higher order logic, linear logic, temporal logic)
- Different from theorems as stated in math
  - theorems in math are informal
  - mathematicians find the formal details too cumbersome

# Human assistance

---

- Some ATPs require human assistance
  - e.g.: programmer gives hints a priori, or interacts with ATP using a prompt
- Hardest theorems to prove are “mathematically interesting” theorems (eg: Fermat’s last theorem)

# Output

---

- Can be as simple as a yes/no answer
- May include proofs and/or counterexamples
- These are formal proofs, not what mathematicians refer to as proofs
- Proofs in math are
  - informal
  - “validated” by peer review
  - meant to convey a message, an intuition of how the proof works -- for this purpose the formal details are too cumbersome

## Output: meaning of the answer

---

- If the theorem prover says “yes” to a formula, what does that tell us?
  - **Soundness**: theorem prover says yes implies formula is correct
  - Subject to bugs in the Trusted Computing Base (TCB)
  - Broad defn of TCB: part the system that must be correct in order to ensure the intended guarantee
  - TCB may include the whole theorem prover
  - Or it may include only a proof checker

## Output: meaning of the answer

---

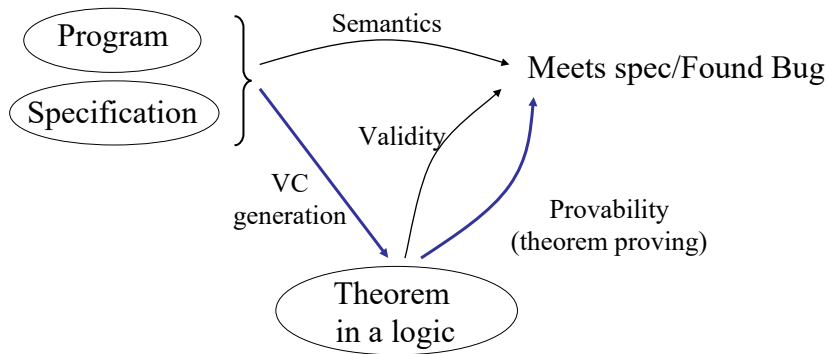
- If the theorem prover says “no” to a formula, what does that tell us?
  - **Completeness:** formula is correct implies theorem prover says yes
  - Or, equivalently, theorem prover says no implies formula incorrect
  - Again, as before, subject to bugs in the TCB

## Output: meaning of the answer

---

- ATPs first strive for soundness, and then for completeness if possible
- Some ATPs are incomplete: “no” answer doesn’t provide any information
- Many subtle variants
  - refutation complete
  - complete semi-algorithm

# Theorem Proving and Software



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

# Programs ! Theorems = Axiomatic Semantics

- Consists of:
  - A language for making assertions about programs
  - Rules for establishing when assertions hold
- Typical assertions:
  - During the execution, only non-null pointers are dereferenced
  - This program terminates with  $x = 0$
- Partial vs. total correctness assertions
  - Safety vs. liveness properties
  - Usually focus on safety (partial correctness)



# Hoare Logic

## C. A. R. (Tony) Hoare

The inventor of this week's logic is also famous for inventing the **Quicksort** algorithm in 1960 - when he was just **26!** A quote:

Computer programming is an **exact science** in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely **deductive reasoning**.



# Hoare Logic

- A way of asserting properties of programs.
- Hoare triple:  $\{A\}P\{B\}$  asserts that “If program  $P$  is started in a state satisfying condition  $A$ , if it terminates, it will terminate in a state satisfying condition  $B$ .”
- A proof system for proving such assertions.
- A way of reasoning about such assertions using the notion of “Weakest Preconditions” (due to Dijkstra).

## A simple programming language

- skip
- $x := e$  (assignment)
- if  $b$  then  $S$  else  $T$  (if-then-else)
- while  $b$  do  $S$  (while)
- $S ; T$  (sequencing)

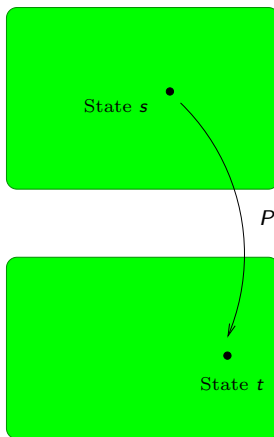
## Example program

```
x := n;  
a := 1;  
while (x ≥ 1) {  
    a := a * x;  
    x := x - 1  
}
```

# Programs as State Transformers

View program  $P$  as a **partial** map  $[P] : \text{Stores} \rightarrow \text{Stores}$ .

All States



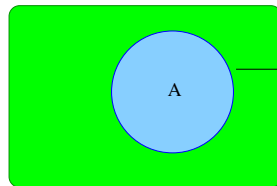
$\{x \mapsto 2, y \mapsto 10, z \mapsto 3\}$

$y = y + 1;$   
 $z = x + y$

$\{x \mapsto 2, y \mapsto 11, z \mapsto 12\}$

# Predicates on States

All States



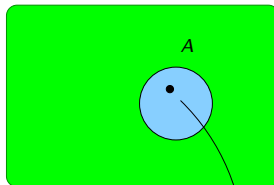
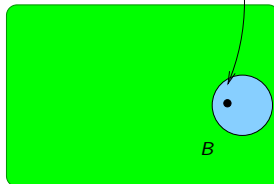
States satisfying  
Predicate A

Eg.  $x \geq 0 \wedge x < y$

Assertion of “Partial Correctness”  $\{A\}P\{B\}$ 

$\{A\}P\{B\}$  asserts that “If program  $P$  is started in a state satisfying condition  $A$ , either it will not terminate, or it will terminate in a state satisfying condition  $B$ .”

All States

 $P$ 

$$\{10 \leq y\}$$

$$\begin{aligned}y &= y + 1; \\ z &= x + y\end{aligned}$$

$$\{x < z\}$$

## Proof rules of Hoare Logic

Skip:

$$\overline{\{A\} \text{ skip } \{A\}}$$

Assignment

$$\overline{\{A[e/x]\} x := e \{A\}}$$



## Proof rules of Hoare Logic

If-then-else:

$$\frac{\{P \wedge b\} S \{Q\}, \{P \wedge \neg b\} T \{Q\}}{\{P\} \text{ if } b \text{ then } S \text{ else } T \{Q\}}$$

While (here  $P$  is called a *loop invariant*)

$$\frac{\{P \wedge b\} S \{P\}}{\{P\} \text{ while } b \text{ do } S \{P \wedge \neg b\}}$$

Sequencing:

$$\frac{\{P\} S \{Q\}, \{Q\} T \{R\}}{\{P\} S; T \{R\}}$$

Weakening:

$$\frac{P \implies Q, \{Q\} S \{R\}, R \implies T}{\{P\} S \{T\}}$$

# Class Activity

- ▶ Download from BB (Week 5) and read the file hoare-logic.pdf (15 minutes to 20 minutes)
- ▶ Discussion

# The KeY Project (Formal Methods for Components and Objects Conf. 2006)

- ▶ The KeY Tool (<https://www.key-project.org>) is a tool for Deductive Verification of Object-Oriented Programs
- ▶ The currently most prominent applications are:
  - ▶ Program Verification (Standalone GUI, Eclipse Integration, KeYHoare)
  - ▶ Debugging (Symbolic Execution Debugger)
  - ▶ Information Flow Analysis / Security
  - ▶ Test Case generation (KeYTestGen)

# Some Buzzwords Early On

- Java as target language
- Dynamic logic as program logic
- Verification = symbolic execution + induction
- Sequent style calculus + meta variables + incremental closure
- Prover is interactive + automated
- Integration with two standard SWE tools:
  - TogetherCC, a commercial CASE tool
  - Eclipse, an open extensible IDE
- Specification languages
  - JML
  - OCL/UML
- Smart cards as main target application

# Supported Specification Languages: OCL

Object Constraint Language

Part of the OMG standard UML

Scope:

Add formal constraints to UML (class) diagrams

# Supported Specification Languages:

## JML

### Java Modeling Language

Behavioral interface specification language for Java

International community effort lead by Gary T. Leavens, Iowa State  
building on the Larch approach

Comes with assertion and runtime checkers

# OCL and JML

## both

- specify method behaviour: pre/post conditions
- specify admissible states: class invariants
- essentially full first order
- support inter-object navigation

## differences

- OCL model oriented:
  - attached to class diagrams
  - 'talks' UML
- JML implementation oriented:
  - attached to Java programs
  - 'talks' Java
  - specifies exceptional behaviour also
- JML only: restricting scope of side effects

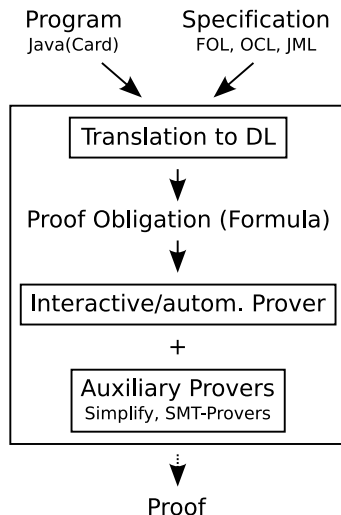
# JML example

```
/*@ public normal_behavior
   @ requires a != null;
   @ ensures (\forall int j; j >= 0 && j < a.length;
              \result >= a[j]);
   @ ensures a.length > 0 ==>
   @       (\exists int j; j >= 0 && j < a.length;
              \result == a[j]);
   @*/

public static /*@ pure @*/ int max(int[] a) {
    if ( a.length == 0 ) return 0;
    int max = a[0], i = 1;
    while ( i < a.length ) {
        if ( a[i] > max ) max = a[i];
        ++i;
    }
    return max;
}
```



# KeY Architecture



# Components of the Calculus

## ① Non-program rules

- first-order rules
- rules for data-types (primarily: arithmetic)
- rules for modalities

## ② Rules for reducing/simplifying the program (symbolic execution)

Replace the program by combination of

- case distinctions (proof branches) and
- sequences of updates

## ③ Rules for handling loops

- rules using loop invariants
- unwinding + induction

## ④ Rules for replacing a method invocations by the method's contract

## ⑤ Update simplification

# Coverage of Java features

## The calculus covers:

- method invocation, dynamic binding
- polymorphism
- abrupt termination
- checking for nullpointer exceptions
- object creation and initialisation
- arrays
- finiteness of integer data types
- transactions (Java Card)

By that, KeY covers the full 'Java Card' language.

# Java Card

- Subset of Java, but with transaction concept
- Sun's official standard for SMART CARDS and embedded devices

## Why Java Card?

Good example for real-world object-oriented language

### Java Card has *no*

- garbage collection
- dynamical class loading
- multi-threading
- floating-point arithmetic

### Application areas

- security critical
- financial risk  
(e.g. exchanging smart cards  
is expensive)

# Implementing Rules: Taclets

## Uniform language for different classes of rules

- First-order calculus
- Specific to Java DL: symbolic execution for Java
- Axioms of theories: arithmetic, lists, etc.
- Lemmas

## Simple, high-level language

- Adding, modifying, and removing formulas
- Conditions restricting applicability of rules
- No complex features like loops
- Suitable both for interactive and automated systems
- Lemmas are validated wrt. base taclets

# Library Case Studies

## Java Collections Framework (JCF)

- Part of JCF (treating sets) specified using UML/OCL
- Some parts of reference implementation verified

## Java Card API

- Most parts of Java Card API specified using UML/OCL
- Some parts of reference implementation verified

## Schorr-Waite Algorithm

- Standard benchmark for verification systems
- Graph marking algorithm for garbage collection
- Java implementation: 2 classes, core algorithm 25 lines of code
- Heavy aliasing, frame problem
- Specified and verified

# Security Case Studies: Java Card Software

## Safety/security properties specified in dynamic logic

- ‘Only certain exceptions can be thrown’
- Transactions are properly used  
(do not commit or abort a transaction that was never started, all started Transactions are also closed)
- Data consistency  
(also if a smartcard is “ripped out” during operation)
- Absence of overflows for integer operations

## Two studies in this area (for which some critical parts were verified)

- Demoney (about 3000 lines):  
Electronic purse application provided by Trusted Logic S.A.
- SafeApplet (about 600 lines): RSA based authentication applet

# Safety Case Study

## Computation of Railway Speed Restrictions

- Software by DBSystems for computing schedules for train drivers: Speed restrictions, required break powers
- Software formally specified using UML/OCL (based on existing informal specification)
- Program translated from Smalltalk to Java

## Avionics Software

- Java implementation of a Flight Manager module at Thales Avionics
- Comprehensive specification using JML, emphasis on class invariants
- Verification of some nested method calls using contracts

## Virtual Machine for Real Time Secury Java

- Verification of some library functions of the Jamaica VM from Aicas