Seminari del corso di laurea in Ingegneria dell'informazione

DIIEIT, Pisa 19 marzo 2011

Introduction to PVS (Prototype Verification System) and logic specifications

Andrea Domenici

DIIEIT, Università di Pisa

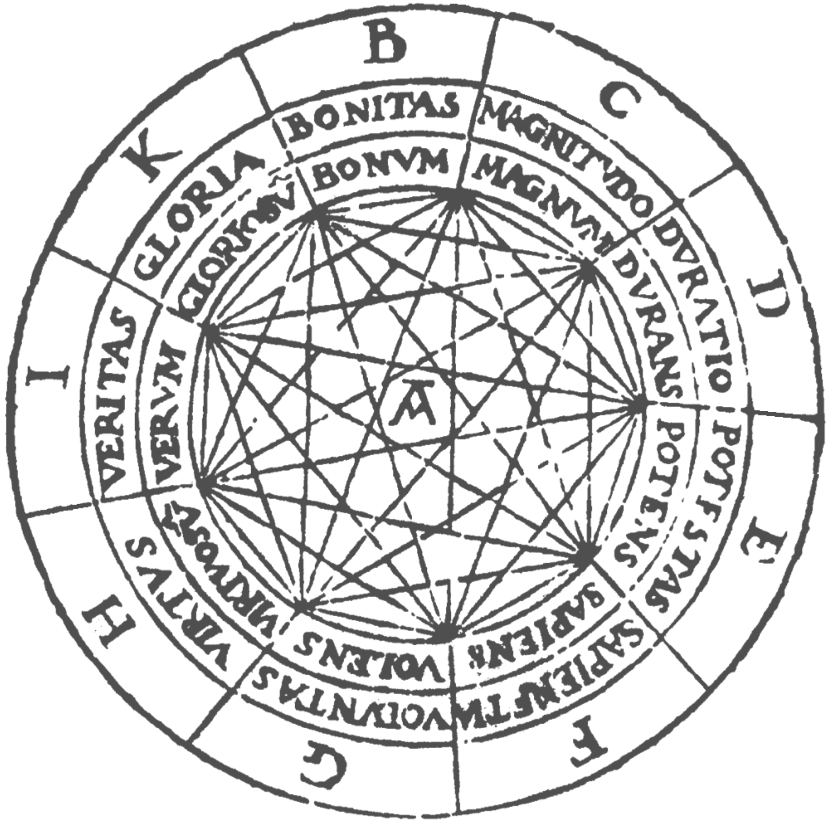
Andrea.Domenici@iet.unipi.it

Outline

- INTRODUCTORY CONCEPTS
  - Formal Logic
  - Languages
  - Formal Systems
  - First-Order Logic
  - Higher-Order Logic
- LOGIC AS A SPECIFICATION LANGUAGE
  - Applying Logic to Technical Problems
  - Formal Languages
  - Theorem Proving
- THE PROTOTYPE VERIFICATION SYSTEM
  - The PVS Specification Language
  - Sequent Calculus and Proofs
  - Prover Commands
- HARDWARE VERIFICATION
  - An Example

INTRODUCTORY CONCEPTS

Raimundus Lullus, doctor illuminatus



Ramon Llull (1232–1315), *Ars magna*.

Formal logic

Sound reasoning and precise language are obviously two indispensable requirements for any scientific and technical activity.

Formal logic is the conceptual framework that explicitly sets out the rules of sound reasoning. Formal logic enables us to make sure that a given line of reasoning (e.g., the demonstration of a theorem) is correct, i.e., the conclusions indeed follow from the premises.

Mathematics and the physical sciences are the classical fields of application for logic, but logic has become an important tool in technical applications, particularly in computer engineering.

Families of logics and formal systems

While the term *logic* refers in general to the science of formal reasoning, we speak of *a logic* or another to refer in particular to some particular way of using the general concepts of logic (just as we have different geometries, Euclidean, Riemannian, etc., within the field of geometry).

There exist several families of logics, with different purposes and expressiveness.

Within each logic, *formal systems* (or *theories*) are defined. Formal systems will be introduced later.

Languages

Wovon man nicht sprechen kann, darüber muß man schweigen.

(Whereof one cannot speak, thereof one must be silent).

L. Wittgenstein, *Tractatus*, Satz No. 7

A logic language defines *what* we want to talk about (the *domain*), *the expressions* that we use to say what we mean (the *syntax*), and how expression are given a meaning (the *semantics*).

- *Domain*: the *individual entities* we talk about, and their reciprocal relationships.
  - E.g., the set of natural numbers, operations, ordering, equality.
- *Syntax*: the *symbols* denoting entities and relationships, and the *well-formedness rules* that say how correct *expressions* can be formed out of symbols. An expression that can be true or false is a (*declarative*) *sentence*, or *formula*.
  - E.g., the symbols $1, 2, 3, \ldots, +, -, \ldots, <, >, =, \ldots$. "$1+1$" is correct, "$1+-2$" is not. "$1<3$" and "$1>3$" are sentences.
- *Semantics*: the rules that relate symbols to entities and relationships, and that decide which sentences are true.

Formal Systems (1)

Given a language and its semantics, we can *interpret* any sentence of the language to see if it is true or false.

E.g., given the sentence "$1 + 1 = 2$", the semantics of the arithmetics language tell us that the symbols "1" and "2" correspond to the concepts of *number one* and *number two*, "$+$" corresponds to *sum*, and "$=$" corresponds to *equality*.

We can then verify (perhaps by counting on our fingers) that the sentence is true.

Things get more complicated when sentences refer to infinite sets, e.g., "*all primes greater than two are odd*"...

Formal Systems (2)

A *formal system* (or *theory*) is a "machine" that we use to *prove* the truth or falsehood of sentences by *deductions*, i.e., by showing that a sentence follows through a series of reasoning steps from some other sentences that are known (or assumed) to be true.

A formal system consists of:

- A *language*;
- a set of *axioms*, selected sentences taken as true[a].
- a set of *inference rules*, saying that a sentence of a given *structure* can be deduced from sentences of the appropriate structure, *independently of the meaning* (*semantics*) of the sentences.
    - E.g., if A and B stand for any two sentences, a well-known inference rule says that from A and "A *implies* B" we can deduce B.

---

[a] Or, more precisely, *valid*.

Formal Systems (3)

More precisely, an inference rule is a relationship between a set of (one or more) formulae called the rule's *premises*, and a formula called the (*direct*) *consequence* of the premises.

E.g., the rule mentioned in the previous slide (the *modus ponens*) is usually written as:

$$\frac{A \quad A \Rightarrow B}{B}$$

or

$$\frac{\begin{array}{c} A \\ A \Rightarrow B \end{array}}{B}$$

Note that this inference rule is a template that is matched by any pair of formulae, since A and B are placeholders for any formula.

Formal Systems (4)

We have a formal system $\mathcal{F}$ with axioms $\mathcal{A}$ and inference rules $\mathcal{R}$

We want to prove that a formula S follows from a set $\mathcal{H}$ of hypotheses.

A *deduction* of S from $\mathcal{H}$ within $\mathcal{F}$ is a sequence of formulae such that S is the last one and each other formula either:

1. Belongs to $\mathcal{A}$; or
2. belongs to $\mathcal{H}$; or
3. is a direct consequence of some preceding formula in the sequence by some rule belonging to $\mathcal{R}$.

The application of an inference rule is a basic step in a formal line of reasoning (or *argument*).

First-Order Logic (1)

A *First-order logic* (FOL) is based on a language consisting of:

- A countable set $\mathcal{C}$ of *constant* symbols, denoting individual entities of the domain;
- a countable set $\mathcal{F}$ of *function* symbols, denoting functions in the domain;
- a countable set $\mathcal{V}$ of *variable* symbols, i.e., placeholders that stand for unspecified *individual entities*;
- a countable set $\mathcal{P}$ of *predicate* symbols, denoting relationships in the domain.
- a finite set of *logical connectives*, e.g. $\neg, \wedge, \vee, \Rightarrow, \ldots$;
- a finite set of *quantifiers*, e.g. $\forall, \exists$.

This is the language we are familiar with from the study of mathematics.

First-Order Logic (2)

A *term* is a constant, a variable, or a function symbol applied, recursively, to an n-tuple of terms.

A term is an expression that denotes an individual entity.

An *atomic formula* (or *atom*) is a predicate symbol applied to an n-tuple of terms.

An atom is an expression whose semantics is *true* iff the entities denoted by its terms satisfy the relationships denoted by the predicate symbol.

A *formula* is an atom, or an expression obtained by combining atoms with quantifiers and connectives.

The semantics of quantifiers and logical connectives are (at least informally) well known, and will not be discussed here.

A Simple First-Order Formal System

- A first-order language with just two connectives ($\neg$ and $\Rightarrow$) and one quantifier ($\forall$);
- The following *axiom schemata*:

$$
\begin{array}{ll}
(1) & A \Rightarrow (B \Rightarrow A) \\
(2) & (A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C)) \\
(3) & (\neg B \Rightarrow \neg A) \Rightarrow ((\neg B \Rightarrow A) \Rightarrow B) \\
(4) & \forall x A(x) \Rightarrow A(t) \\
(5) & \forall x (A \Rightarrow B) \Rightarrow (A \Rightarrow \forall x B)
\end{array}
$$

- The following rules of inference:

$$\frac{A \quad A \Rightarrow B}{B}$$

$$\frac{\mathcal{A}}{\forall x \mathcal{A}}$$

In the second rule (*generalization*), there are constraints on x.

Higher-Order Logic

In a FOL, variables may range only over individual entities.

In a FOL, we may say "*For all x's such that x is a real number,* $x^2 = x \cdot x$".

We *cannot* say "*For all f's such that f is a function over real numbers,* $f^2(x) = f(x) \cdot f(x)$".

In *higher-order* logics, variables may range over functions and predicates.

In higher-order logics, we can make statements about predicates: e.g., we may say "*if x and y are real numbers and $x = y$, then for all P's such that P is a predicate,* $P(x) = P(y)$".

Applying logic to technical problems

Formal logic is used in mathematics to investigate properties of abstract concepts, such as geometrical shapes, numbers, functions. . .

However, it can be used to describe and reason about technical systems, such as computer programs, electronic circuits, industrial control systems. . .

A formal system enables developers to:

- Describe system characteristics and requirements with great rigor and accurateness;
- formally prove system properties.

A *great* number of formal systems have been devised for requirements specification and system verification.

Formal languages (1)

- A formal language identifies some basic attributes that are simple and general enough to describe a large class of systems in an abstract way.
  - E.g., the behavior of many systems can be described in terms of sets of *states* and sequences of *actions*.
- The possible values of these attributes form the domain of the language (just like numbers form the domain of algebra).
- The language defines operations that act on the elements of the domain, such as forming sets and sequences, and combining them in various ways.
  - E.g., we may define operations for *parallel* and *sequential* composition to describe the interaction of two processes.
- We can then describe systems with formulae whose meaning can be understood in terms of mathematical concepts, such as sets and functions.

Formal languages (2)

Some families of logic-based specification languages:

- *Predicate logics*. Based on predicate logic and set theory, very general applicability.
- *Temporal logics*. Used to specify properties related to synchronization.
- *Process algebras*. A large class of languages that describe concurrent processes by means of operators on elementary actions. Often used in conjunction with temporal logics.
- . . .

A few modeling languages

- *Z* (/zεd/). Based on predicate logic and Zermelo-Fränkel set theory.
- *Vienna Development Method* (VDM). Well-known predicate logic formalism.
- *Prototype Verification System* (PVS). More about this later on. . .
- *Calculus of Communicating Systems* (CCS). A process algebra.
- *Communicating Sequential Processes* (CSP). Another process algebra.
- *Language of Temporal Ordering Specification* (LOTOS). Yet another process algebra.
- . . .

Can Properties Be Verified Mechanically?

No. Well, sometimes yes.

In a previous slide, we described a formal system as a “machine” to prove truth or falsehood of sentences by the process of deduction.

However, such a machine does not run by itself. Proving a formula is much like a game where one must choose the right moves (inference steps) and do them in the right order.

Many proof *strategies* exist to guide deduction, such as *proof by induction* or *proof by contradiction*.

In general, *no proof strategy may be guaranteed to prove or disprove an arbitrary formula* in a given formal system (problem of *decidability*).

However, there are classes of formulae that are decidable. *In such cases, it is possible to use a mechanical procedure*.

Theorem Proving and Model Checking

Two main aproaches exist to automatic verification of system properties:

- *Theorem proving*: A *theorem prover* is a computer program that implements a formal system. It takes as input a formal definition of the system that must be verified and of the properties that must be proved, and tries to construct a proof by application of inference rules, according to a built-in strategy.
- *Model checking*: A *model checker* is a computer program that extracts a *model* of the system to be verified from its formal description. The model is a graph whose nodes are the states of the system, connected by transitions. The model checker examines each state and checks if the desired properties hold in that state.

Theorem proving may be fully *automatic*, or *interactive*.

Prototype Verification System

The PVS is an interactive theorem prover developed at Computer Science Laboratory, SRI International, Menlo Park (California), by S. Owre, N. Shankar, J. Rushby, and others.

The formal system of PVS consists of a higher-order language and the *sequent calculus* axioms and inference rules.

PVS has many applications, including formal verification of hardware, algorithms, real-time and safety-critical systems.

Using the PVS

- EMACS-based user interface.
- The user writes definitions and formulae.
- The user selects a formula and enters the *prover* environment.
- Prover commands apply single inference rules or pre-packaged sequences of rules (*strategies*), transforming formulae or producing new formulae.
- The user examines the formulae resulting for each prover command, and decides what to do next.
- The prover finds out when a proof has been successfully completed.

The PVS Specification Language

- Logical connectives: `NOT`, `AND`, `OR`, `IMPLIES`, ...
- Quantifiers: `EXISTS`, `FORALL`.
- Complex operators: `IF-THEN-ELSE`, `COND`.
- Notation for records, tuples, lists...
- Notation for definitions, abbreviations...
- Rich higher-order type system. Each variable is defined to range over a type, including function and predicate types (predicates are functions that return a Boolean value).
- *Theories*: named collections of definitions and formulae. A theory may be *imported* (and referred to) by another theory.
- A large number of pre-defined theories is available in the *prelude* library.

Typed Logic

- Every variable or constant belongs to a type, i.e., denotes elements of a given set.
- *Pre-defined base types*: `bool`, `nat`, `real`...
- *Uninterpreted types*: we just say that a type with a given name exists, e.g., `perfectsw: TYPE`.
- *Interpreted types*: we define a type in terms of other types, or by explicit enumeration of its members.
  - *Enumerations*: `flag: TYPE = {red, black, white, green}`
  - *Tuples*: `triple: TYPE = [nat, flag, real]`
  - *Records*: `point: TYPE = [# x: real, y: real #]`
  - *Subtypes*: `posnat: TYPE = {x: nat | x>0}`
  - *Functions*: `int2int: TYPE = [int -> int]`

Declarations

- *Constants*:
  - `n0: nat` (*uninterpreted constant*)
  - `lucky: nat = 13`
  - `a_triple: triple = (lucky, red, 3.14)`
  - `origin: point = (# x := 0.0, y:= 0.0 #)`
  - `inc: int2int = (lambda (x: int): x + 1)`
  - `inc: [int -> int] = (lambda (x: int): x + 1)`
  - `inc(x: int): int = x + 1`
- *Variables*: add `VAR` to type expression: `m: VAR nat`
- *Formulae*:
  - `plus_commutativity: AXIOM forall(x, y: nat): x + y = y + x`
  - `a_theorem: THEOREM forall(n: nat): n < n + 1`

Keyword `lambda` introduces the parameters of a function.

Instead of `THEOREM` we may use `LEMMA`, `CONJECTURE`. . .

An `AXIOM` is assumed to be proved.

Example: Groups

```
group : THEORY
BEGIN
  G : TYPE+      % uninterpreted, nonempty
  e : G          % neutral element
  i : [G -> G]   % inverse
  * : [G,G -> G] % binary operation
  x,y,z : VAR G
  associative : AXIOM
    (x * y) * z = x * (y * z)
  id_left : AXIOM
    e * x = x
  inverse_left : AXIOM
    i(x) * x = e
  inverse_associative : THEOREM
    i(x) * (x * y) = y
END group
```

Sequent calculus (1)

Gerhard Gentzen (1909, 1945).

Sequent calculus (2)

The sequent calculus works on formulae of a special form, called *sequents*, such as:

$$A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m$$

where the A_i's are the *antecedents* and the B_i's are the *consequents*.

Each antecedent or consequent, in turn, is a formula of any form (it may contain subformulae with quantifiers and connectives, but not "sub-sequents").

The symbol in the middle ($\vdash$) is called a *turnstile* and may be read as "*yields*".

Informally, a sequent can be seen as another notation for

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n \Rightarrow B_1 \vee B_2 \vee \ldots \vee B_m$$

Sequent calculus (3)

A sequent is true if:

- Any formula occurs both as an antecedent and as a consequent; or
- any antecedent is false; or
- any consequent is true.

In the PVS prover interface, a sequent is represented as:

```
{-1}  A1
 ...
[-n]  An
  |-------
{1}   B1
 ...
[m]   Bm
```

Sequent calculus (4)

The Sequent calculus has one axiom: $\Gamma, A \vdash A, \Delta$ where Γ and Δ are (multi)sets of formulae.

Inference rules:

$$\frac{}{\Gamma,A\vdash A,\Delta}\,\mathrm{axm} \qquad \frac{\Gamma\vdash\Delta,A \quad A,\Gamma\vdash\Delta}{\Gamma\vdash\Delta}\ \mathrm{cut} \qquad \frac{A,A,\Gamma\vdash\Delta}{A,\Gamma\vdash\Delta}\,\mathrm{ctr\ L} \qquad \frac{\Gamma\vdash\Delta,A,A}{\Gamma\vdash\Delta,A}\,\mathrm{ctr\ R}$$

$$\frac{\Gamma\vdash\Delta,A}{\neg A,\Gamma\vdash\Delta}\neg L \qquad \frac{A,\Gamma\vdash\Delta}{\Gamma\vdash\Delta,\neg A}\neg R \qquad \frac{A,B,\Gamma\vdash\Delta}{A\wedge B,\Gamma\vdash\Delta}\wedge L \qquad \frac{\Gamma\vdash\Delta,A \quad \Gamma\vdash\Delta,B}{\Gamma\vdash\Delta,A\wedge B}\wedge R$$

$$\frac{A,\Gamma\vdash\Delta \quad B,\Gamma\vdash\Delta}{A\vee B,\Gamma\vdash\Delta}\vee L \qquad \frac{\Gamma\vdash\Delta,A,B}{\Gamma\vdash\Delta,A\vee B}\vee R \qquad \frac{\Gamma\vdash\Delta,A \quad B,\Gamma\vdash\Delta}{A\Rightarrow B,\Gamma\vdash\Delta}\Rightarrow L \qquad \frac{A,\Gamma\vdash\Delta,B}{\Gamma\vdash A\Rightarrow B,\Delta}\Rightarrow R$$

$$\frac{A[x\leftarrow t],\Gamma\vdash\Delta}{\forall x.A,\Gamma\vdash\Delta}\forall L \qquad \frac{\Gamma\vdash\Delta,A[x\leftarrow y]}{\Gamma\vdash\forall x.A,\Delta}\forall R \qquad \frac{A[x\leftarrow y],\Gamma\vdash\Delta}{\exists x.A,\Gamma\vdash\Delta}\exists L \qquad \frac{\Gamma\vdash\Delta,A[x\leftarrow t]}{\Gamma\vdash\exists x.A,\Delta}\exists R$$

axm: the axiom
cut: the cut rule
ctr: the contraction rules

The quantifier rules have caveats on the quantified variable.

Proofs

Proofs are constructed backwards from the *goal* sequent, that in PVS has the form $\vdash F$, where F is the formula we want to prove.

Inference rules are applied backwards, i.e., given a formula, we find a rule whose consequence matches the formula, and the premises become the new *subgoals*.

Since a rule may have two premises, proving a goal produces a tree of sequents, rooted in the goal, called the *proof tree*.

The proof is completed when (and if!) all branches terminate with an instance of the axiom.

Proof Example

Suppose we want to prove that $\neg A \vee \neg B \Rightarrow \neg(A \wedge B)$.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{A, B \vdash A}\ \text{axm}}{\neg A, A, B \vdash}\ \neg L \qquad \dfrac{\overline{A, B \vdash B}\ \text{axm}}{\neg B, A, B \vdash}\ \neg L}{(\neg A \vee \neg B), A, B \vdash}\ \vee L}{(\neg A \vee \neg B), (A \wedge B) \vdash}\ \wedge L}{(\neg A \vee \neg B) \vdash \neg(A \wedge B)}\ \neg R}{\vdash (\neg A \vee \neg B) \Rightarrow \neg(A \wedge B)}\ \Rightarrow R
$$

The root goal is at the bottom.

At the top we have two branches that end with empty formulae by the axiom rule.

The goal has then been proved.

Prover Commands

The PVS prover has a large number of commands (also called *rules*):

- *Control* rules to control proof execution and proof tree exploration.
- *Structural* rules to implement the contraction rules and to hide unused formulae in the sequent.
- *Propositional* rules implement the inference rules for connectives, for complex operators, and for the cut. They also apply various simplification laws.
- *Quantifier* rules implement the inference rules for quantifiers.
- *Equality* rules implement various inference rules, including rules for equality, records, tuples, and function definitions.
- *Definition and lemma handling* rules invoke and apply lemmas and definitions.
- *Strategies* apply pre-defined sequences of rules.
- …and more.

Prover Commands: `flatten`

`flatten` implements the $\wedge L$, $\vee R$, and $\Rightarrow R$ rules:

```
{-1} A AND B
  |------
{1} C OR D
{2} E => F

Rule? (flatten)

{-1} A
{-2} B
{-3} E
  |------
{1} C
{2} D
{3} F
```

Prover Commands: `split`

`split` implements the $\wedge R$, $\vee L$, and $\Rightarrow\ L$ rules:

```
  |------
{1} A AND B

Rule? (split)

  |------              |------
{1} A               {1} B
```

The `split` command produced two subgoals, i.e., a branching point in the proof tree.

Prover Commands: `skolem`

`skolem` implements the $\forall R$ and $\exists L$ rules:

```
  |------
{1} FORALL (x:T): P(x)

Rule? (skolem 1 "c")

  |------
{1} P(c)
```

```
{-1} EXISTS (x:T): P(x)
  |------
{1} A

Rule? (skolem -1 "c")

{-1} P(c)
  |------
{1} A
```

Prover Commands: `skosimp*`

`skosimp*` is a strategy that applies skolemization and `flatten`:

```
  |------
{1} FORALL (x:T): P(x) => FORALL (y:S): Q(y) OR R(y)

Rule? (skosimp*)

{-1} P(x!1)
  |------
{1} Q(y!1)
{2} R(y!1)
```

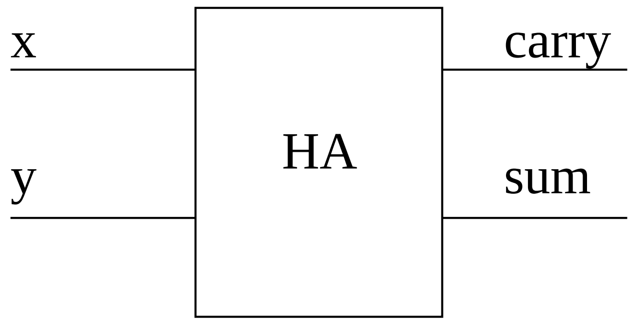
Often the first step in a proof.

HARDWARE VERIFICATION

Example: a Half Adder

Specification:

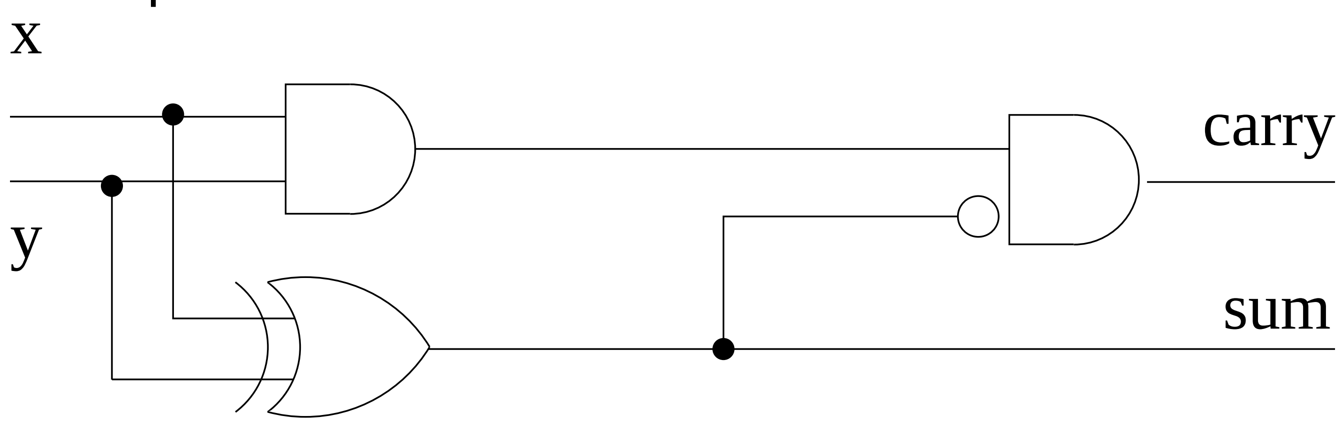
x	y	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Is the implementation

correct?

An implementation:



A mathematical specification

A half adder is a device whose input is the 1-digit binary encoding of two natural numbers, and whose output is the 2-digit binary encoding of their sum.

A half adder must correctly execute the sums $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, and $1 + 1 = 2$.

In PVS, the property of correctness for the half adder can be expressed as:

```
2*b2n(carry) + b2n(sum)
        = b2n(x) + b2n(y)
```

where `b2n` is a function that translates a Boolean value into a natural number in $\{0, 1\}$.

Half Adder Theory

```
HA : THEORY
BEGIN
  x,y : VAR bool

  HA(x,y) : [bool, bool] =
     ((x AND y) AND (NOT (x XOR y)),        % carry
      (x XOR y))                            % sum

  % convert Boolean to natural
  b2n(x) : nat = IF x THEN 1 ELSE 0 ENDIF

  HA_corr : THEOREM        % correctness
    LET (carry, sum) = HA(x, y) IN
      b2n(sum) + 2*b2n(carry)
        = b2n(x) + b2n(y)
END HA
```

A Proof (1)

Initial goal:

```
HA_corr :

  |-------
{1}   FORALL (x, y: bool):
        LET (carry, sum) = HA(x, y) IN
          b2n(sum) + 2 * b2n(carry)
            = b2n(x) + b2n(y)
```

Get rid of quantifiers:

```
Rule? (skolem*)
HA_corr :

  |-------
{1}   LET (carry, sum) = HA(x!1, y!1) IN
        b2n(sum) + 2 * b2n(carry)
            = b2n(x!1) + b2n(y!1)
```

A Proof (2)

Get rid of let-expressions:

```
Rule? (beta)
HA_corr :

  |-------
{1}   b2n(HA(x!1, y!1)‘2) + 2 * b2n(HA(x!1, y!1)‘1) =
       b2n(x!1) + b2n(y!1)
```

Expand definition of b2n:

```
Rule? (expand "b2n")
HA_corr :

  |-------
{1}   IF HA(x!1, y!1)‘2 THEN 1 ELSE 0 ENDIF +
       2 * IF HA(x!1, y!1)‘1 THEN 1 ELSE 0 ENDIF
       = IF x!1 THEN 1 ELSE 0 ENDIF
           + IF y!1 THEN 1 ELSE 0 ENDIF
```

A Proof (3)

Factor out conditionals:

```
Rule? (lift-if)
HA_corr :

  |-------
{1}   IF HA(x!1, y!1)‘2
      THEN 1
           + 2 * IF HA(x!1, y!1)‘1 THEN 1 ELSE 0 ENDIF
          = IF x!1 THEN 1 ELSE 0 ENDIF
           + IF y!1 THEN 1 ELSE 0 ENDIF
      ELSE 0
           + 2 * IF HA(x!1, y!1)‘1 THEN 1 ELSE 0 ENDIF
          = IF x!1 THEN 1 ELSE 0 ENDIF
           + IF y!1 THEN 1 ELSE 0 ENDIF
      ENDIF
```

A Proof (4)

After many lift-if's, expand HA:

```
Rule? (expand "HA")
HA_corr :

  |-------
{1}   IF (x!1 XOR y!1)
      THEN IF x!1 THEN IF y!1 THEN FALSE ELSE TRUE ENDIF
             ELSE IF y!1 THEN TRUE ELSE FALSE ENDIF
             ENDIF
      ELSE IF (x!1 AND y!1) THEN TRUE
           ELSE IF x!1 THEN FALSE
                ELSE IF y!1 THEN FALSE ELSE TRUE ENDIF
                ENDIF
           ENDIF
      ENDIF
```

A Proof (5)

Boring Boolean algebra:

```
Rule? (prop)
this yields  4 subgoals:
HA_corr.1 :

{-1}  y!1
{-2}  x!1
{-3}  (x!1 XOR y!1)
  |-------
```

The goal branches into four subgoals, `HA_corr.1` through `HA_corr.4`.

A Proof (6)

Grind:

```
Rule? (grind)
Trying repeated skolemization,
instantiation, and if-lifting,

This completes the proof of HA_corr.1.
```

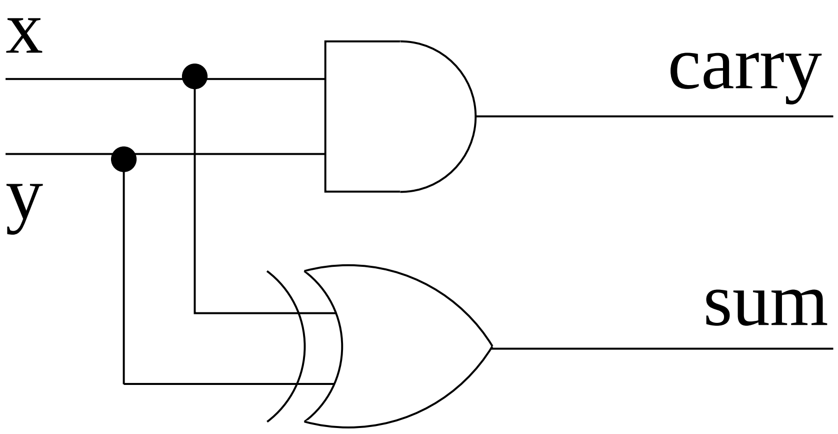
And so on until:

```
Rule? (grind)
Trying repeated skolemization,
instantiation, and if-lifting,

This completes the proof of HA_corr.4.

Q.E.D.
```

Another implementation




```
HA2(x,y) : [bool, bool] =
   ((x AND y),             % carry
    (x XOR y))             % sum
```

Another implementation

The correctness theorem has the same form as before, only the half adder function changes:

```
HA2_corr : THEOREM       % correctness
  LET (carry, sum) = HA2(x, y) IN
    bool2nat(sum) + 2*bool2nat(carry)
      = bool2nat(x) + bool2nat(y)
```

We can also prove that the two implementations are equivalent:

```
HA_HA2_equiv: THEOREM    % equivalence
    HA(x, y) = HA2(x, y)
```

Reusing a Proof (1)

To prove correctness for the new implementation (`HA2`), we reuse the proof for `HA`, using the `install-proof` prover command.

The old proof is rerun automatically until the commands are exhausted or no longer applicable, then the user is prompted:

```
HA2_corr.5 :

{-1}  y!1
{-2}  x!1
{-3}  HA2(x!1, y!1)‘2
  |-------
{1}   1 + 2 * 0 = 1 + 1
{2}   HA2(x!1, y!1)‘1

Rule?
```

A Proof (2)

We expand `HA2` and we complete the proof with a few applications of `grind`.

Proving the equivalence of `HA` and `HA2` is trivial:

```
HA_HA2_equiv :

  |-------
1   FORALL (x, y: bool): HA(x, y) = HA2(x, y)

Rule? (grind)
  ...
Trying repeated skolemization, instantiation, and if-lifting
Q.E.D.
```

Actually, all these proofs can be done with a single `grind` step.
But. . .

Do not overestimate the power of Grind

$\longrightarrow$

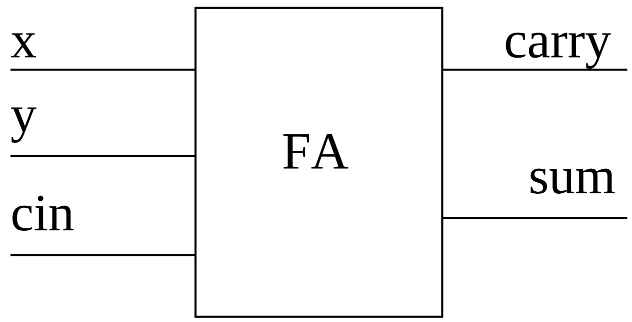
In any realistic problem, the proof requires an intelligent choice of axioms and inference steps.

The `grind` strategy is useful in the last steps.

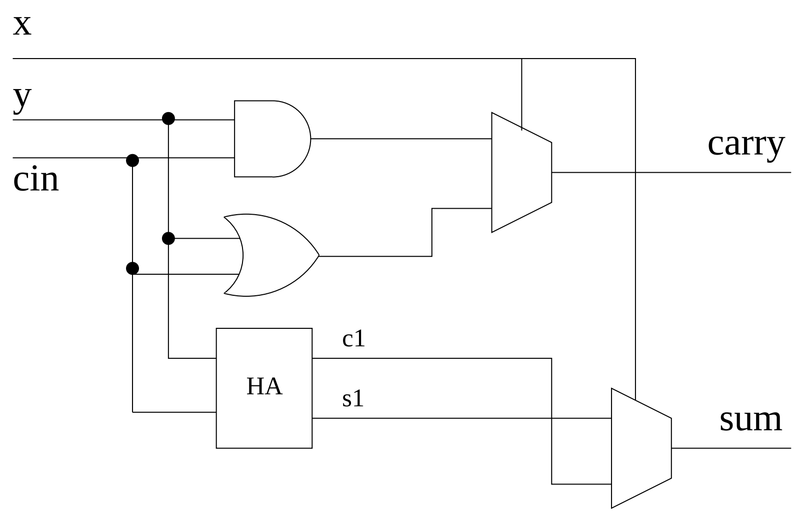
Using `grind` too early may produce a large and messy amount of hard-to-read subgoals, and even make the proof impossible.

Example: a Full Adder

x	y	cin	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



An implementation:



Reusing a Theory

```
FA2 : THEORY
BEGIN
  importing HA                % theory composition

  % 2-to-1 multiplexer
  mux2(x: bool, y: bool, s: bool): bool =
     if s then y else x endif

  FA2(x: bool, y: bool, cin: bool) : [bool, bool] =
     LET (c1, s1) = HA2(y, cin) IN
     (mux2(y and cin, y or cin, x),   % carry
      mux2(s1, not s1, x))            % sum

  FA2_corr : THEOREM                  % correctness
    LET (carry, sum) = FA2(x, y, cin) IN
      b2n(sum) + 2 * b2n(carry)
        = b2n(x) + b2n(y) + b2n(cin)
END FA2
```

Conclusions

We have taken a first look at an interesting tool for the formal verification and validation of specifications and designs.

We have (almost) totally ignored the theoretical aspects.

The focus was on giving an idea of the tool's features and possibilities.

The philosophy behind this tool might be expressed in this way:

> *Assisting human insight with the power and reliability of mechanical theorem proving*.

Of course there is a lot more to it: please go through the references, and Google be with you! You will find *lots* of interesting stuff.

Acknowledgements

Warm thanks to Cinzia Bernardeschi (University of Pisa), Paolo Masci (Queen Mary University of London), and Holger Pfeifer (Technische Universität München), who introduced me to the PVS.

I am particularly indebted to the latter, as most of the material in this seminar is based on *his* seminars, part of which I have shamelessly copied.

Some References

[1] S. Owre, J. Rushby, N. Shankar, and F. von Henke, “Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS,” IEEE Trans. on Software Engineering, vol. 21, no. 2, pp. 107–125, 1995.

[2] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas, “PVS: combining specification, proof checking, and model checking,” in Computer-Aided Verification, CAV ’96, ser. LNCS, R. Alur and T. Henzinger, Eds. Springer-Verlag, 1996, no. 1102, pp. 411–414.

[3] S. Owre, J. Rushby, N. Shankar, and M. Srivas, “A tutorial on using PVS for hardware verification,” in Theorem Provers in Circuit Design (TPCD ’94), ser. LNCS, R. Kumar and T. Kropf, Eds. Springer-Verlag, 1997, no. 901, pp. 258–279.

[4] M. Srivas, H. Rueß, and D. Cyrluk, “Hardware verification using PVS,” in Formal Hardware Verification: Methods and Systems in Comparison, ser. LNCS, T. Kropf, Ed. Springer-Verlag, 1997, no. 1287, pp. 156–205.

Some References

[5] S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert, “PVS: an experience report,” in Applied Formal Methods, ser. LNCS. Springer-Verlag, 1998, no. 531, pp. 338–345.

[6] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert, “Eval-uating, testing, and animating PVS specifications,” Computer Science Laboratory, SRI International, Tech. Rep., 2001.

[7] C. Muñoz, “Rapid prototyping in PVS,” National Institute of Aerospace, Hampton, VA, USA, Tech. Rep. NIA 2003-03, NASA/CR-2003-212418, 2003.

Thank you

شُكرًا