Lu Yan

Formal Verification of a Ubiquitous Hardware Component

Turku Centre *for* Computer Science

TUCS Technical Report
No 637, November 2004

Formal Verification of a Ubiquitous Hardware Component

Lu Yan
Turku Centre for Computer Science
Lemminkäisenkatu 14, FI-20520 Turku, Finland
lyan@abo.fi

TUCS Technical Report
No 637, November 2004

Abstract

The paper begins by discussing various approaches to hardware specification and verification. The main emphasis is on using mechanical verification tools to assist the verification process. The case study is the verification of a seven-segment LED display decoder circuit design, in which two popular verification tools, HOL and PVS, are compared and evaluated.

Keywords: Hardware verification, formal methods, ubiquitous systems

TUCS Laboratory
Distributed Systems Design Laboratory

1 Introduction

The development of microelectronics has allowed hardware designers to build remarkably complex devices. However, it becomes increasingly difficult to ensure these devices free of design errors. In most cases, exhaustive simulation of a medium size design is impossible and the correctness of the design cannot be assured. This is a serious problem in safety-critical applications, where a small design error may cause loss of life and extensive damage. Even in the case where safety is not the primary concern, a design error means costly and time-consuming rechecking in massive production lines.

A solution to the problem is to apply formal methods for verification of correctness of hardware designs - hardware verification. With this approach, the behavior of hardware is described mathematically, and formal proof is used to verify the intended behavior. The proofs can be very large and complex, so mechanical verification tools are often used to assist the verification.

We illustrate our experiences with formal verification in ubiquitous hardware design via a comparative case study of the verification of a circuit design of seven-segment LED display decoder: A seven-segment LED display is comprised of seven light emitting diodes (LED). Input signals are applied to the input port of the seven-segment decoder, and the decoder translates them into ON/OFF status of the seven LEDs. Then, selected combinations of the LEDs are illuminated to display numeric digits and other symbols.

2 What is formal hardware verification

We consider a formal hardware verification problem to consist of *formally establishing that an implementation satisfies a specification.* The term *implementation (Imp)* refers to the hardware design that is to be verified. This entity can correspond to a design description at any level of the hardware abstraction hierarchy, not just the final physical layout (as is traditionally regarded in some areas). The term *specification (Spec)* refers to the property with respect to which correctness is to be determined. It can be expressed in a variety of ways - as a behavioral description, as an abstracted structural description, as a timing requirement etc.

In particular, we do not address directly the problem of specification validation, i.e. whether the specification means what it is intended to mean, whether it really expresses the property one desires to verify, whether it completely characterizes correct operation etc. A specification for a particular verification problem can itself be made the object of scrutiny, by serving as an implementation for another verification problem at a conceptually higher level. Similarly, at the lowest end too, we do not specifically address the problem of model validation, i.e. whether the model used to represent the implementation is consistent, valid, correct etc. It is obvious that the quality of verification can only be as good as the quality of the models used.

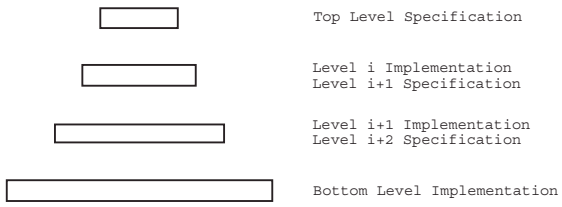


Figure 1: Hierarchical verification[1]

An important feature of the above formulation is that it admits hierarchical verification corresponding to successive levels of the hardware abstraction hierarchy. Typically, the design of a hardware system is organized at different levels of abstraction, the topmost level representing the most abstract view of the system and the bottommost being the least abstract, usually consisting of actual layouts. Verification tasks can also be organized naturally at these same levels. An implementation description for a task at any given level, serves also as a statement of the specification for a task at the next lower level, as shown in Figure 1. In this manner, top-level specifications can be successively implemented and verified at each level, thus leading to implementation of an overall verified system. Hierarchical organization not only makes this verification process natural, it also makes the task tractable. By breaking this large problem into smaller pieces that can be handled individually, the verification problem is made more manageable. It effectively increases the range of circuit sizes that can be handled in practice.

2.1 Hardware verification method

Two things are needed for any method of hardware verification based on rigorous specification and formal proof. The first is a formal language for describing the behaviors of hardware and expressing proposition about it. The ideal language is expressive enough to describe hardware in a natural concise notation yet still has a well-understood and reasonably simple semantic. The second requirement is a deductive calculus for proving propositions expressed in this language. It must be logically sound and it should be powerful enough to allow one to prove all the true propositions about hardware behavior that arise in practice.

Various formal languages and associated proof techniques have been proposed as a basis for hardware verification. These range from special-purpose hardware description languages with *ad hoc* proof rules to systems of formal logic and subsets of ordinary mathematics. Formal methods for reasoning about hardware behavior have been based, for example, on algebraic techniques, various kinds of temporal logic, functional programming techniques, predicate calculus, and higher order logic.

The details of the verification methods based on these different formalisms vary, but many of them share a common general approach. This typically involves the following four steps:

1. Write a formal specification S to describe the behavior that the device to be verified must exhibit for it to be considered correct.

2. Write a specification for each kind of primitive hardware component used in the device. These specifications are intended to describe the actual behavior of real hardware components.

3. Define an expression D which describes the behavior of the device to be proved correct. The definition of D has the general form

$$D = P_1 + \cdots + P_n$$

   where $P_1, \cdots, Pn$ specify the behavior of the constituent parts of the device and $+$ is a composition operator which models the effect of wiring components together. The expressions $P_1, \cdots, Pn$ used here are instances of the specifications for primitive devices defined in step 2.

4. Prove that the device described by the expression D is correct with respect to the specification S. This is done by proving a theorem of the form

$$\vdash D \quad \text{satisfies} \quad S$$

   where 'satisfies' is some satisfaction relation on specifications of hardware behavior. This correctness theorem asserts that the behavior described by D satisfies the specification of intended behavior S.

When the device to be proved correct is large, this method is usually applied hierarchically. The design is structured into a hierarchy of components and subcomponents, and specifications that describe primitive components at one level of the hierarchy then become specifications of intended behavior at the next level down. The structure of the proof mirrors this hierarchy: the top-level specification is shown to be satisfied by an appropriate connection of components; at the next level down, each of these components is shown to be correctly implemented by a connection of sub-components, and so on down to the lowest level, where the components used correspond to devices available as hardware primitives.[31]

2.2 Hardware verification using higher order logic

The version of higher order logic described here was developed by Mike Gordon at the University of Cambridge. The main difference between first order logic and higher order logic is that higher order logic allows quantification over predicates. The ability to quantify over predicate symbols leads to a greater power of expressiveness in higher order logic. Another significant difference is that higher order logic admits higher order predicates and functions, i.e. arguments and results of

predicates and functions can themselves be predicates or functions. This allows functions to be manipulated just like ordinary values, which leads to a more mathematically elegant formalism.

The following short description of higher order logic is not complete, although it covers important notations of the logic, which provides some background information for the later example. A full description of higher order logic can be found at [17].

Types Higher order logic is a typed logic. The syntax of types in higher order logic is given by

$$\sigma ::= c|v|(\sigma_1, \ldots, \sigma_n)op$$

where $\sigma, \sigma_1, \ldots, \sigma_n$ range over types, c ranges over type constants, v ranges over type variables, and op ranges over n-ary type operators.

Terms The notation of terms in higher order logic can be viewed informally as an extension of the conventional syntax of predicate calculus in which variables can range over functions and functions can take functions as arguments or yield functions as results. The syntax of terms in higher order logic is given by

$$M ::= c|v|(MN)|\lambda v.M$$

where c ranges over constants, v ranges over variables, and M and N range over terms.

Sequents, theorems and inference rules A sequent is written $\Gamma \vdash P$, where Γ is a set of boolean terms called assumptions and P is a boolean term called the conclusion. When the set Γ is empty, the notation $\vdash P$ is used. In this case, P is a formal theorem of the logic. The same notation is used for the axioms of the logic. All inference rules of the logic can be found at Table 1.

The approach to specifying hardware behavior in higher order logic is to specify the behavior of a device by describing the combinations of values that can be observed on its external wires. A specification is expressed formally in logic by a boolean-valued term whose free variables correspond to these external wires. This term imposes a constraint on the values of these variables. To reflect the behavior of the device it specifies, the term is chosen so that the combinations of values that satisfy this constraint are precisely those which can be observed simultaneously on the corresponding external wires of the device itself.

As an example, consider the device **Dev** shown below.

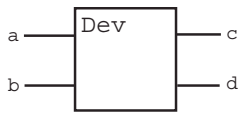


Table 1: Inference rules of higher order logic

1. ASSUME: $\frac{}{\{P\}\vdash P}$
2. REFL: $\frac{}{\vdash N=N}$
3. BETA_CONV: $\frac{}{\vdash(\lambda v.N)M=N[M/v]}$
4. ABS: $\frac{\Gamma\vdash M=N}{\Gamma\vdash(\lambda v.M)=(\lambda v.N)}$ (v not free in Γ)
5. INST_TYPE: $\frac{\Gamma\vdash P}{\Gamma\vdash P[\sigma_1,\ldots,\sigma_n/\alpha_1,\ldots,\alpha_n]}$
6. DISCH: $\frac{\Gamma\vdash P}{\Gamma-\{Q\}\vdash Q\supset P}$
7. MP: $\frac{\Gamma_1\vdash P\supset Q \quad \Gamma_2\vdash P}{\Gamma_1\cup\Gamma_2\vdash Q}$
8. SUBST: $\frac{\Gamma_1\vdash N'_1 \quad \ldots \quad \Gamma_n\vdash N_n=N'_n \quad \Gamma\vdash P}{\Gamma_1\cup\ldots\cup\Gamma_n\cup\Gamma\vdash P[N'_1,\ldots,N'_n/N_1,\ldots,N_n]}$

This device has four external wires: a, b, c, and d. A specification of its behavior in logic is therefore a boolean-valued term of the form $S[a, b, c, d]$, constructed so that for all values of the free variables a, b, c and d:

$$S[a,b,c,d] = \begin{cases} \mathbf{T} & \text{if the values } a, b, c, \text{ and } d \text{ could occur} \\ & \text{simultaneously on the corresponding} \\ & \text{external wires of the device } \mathbf{Dev} \\ \mathbf{F} & \text{otherwise} \end{cases}$$

This approach to specifying hardware describes its behavior only in terms of the values that can be observed externally. No information about internal state is used in a specification. Furthermore, there is no distinction between the inputs and the outputs of a device; the constraint imposed by a specification on its free variables need not be a functional one. Of course, the free variables in a specification need not stand for the values on the physical wires of an actual circuit; they may represent more abstract externally observable quantities. Both specifications of hardware primitives and specifications of the intended behavior of designs can therefore be expressed by this method.

Once a design has been constructed, its correctness can be expressed by a proposition which asserts that this design in some sense satisfies an appropriate specification of required or intended behavior. The most direct way of formulating this satisfaction of a design is asserted by a theorem of the form

$$\vdash D[v_1,\ldots,v_n] = S[v_1,\ldots,v_n],$$

Where the term $D[v_1, \ldots, v_n]$ is the design of the device asserted to be correct and the term $S[v_1, \ldots, v_n]$ is the specification of required behavior. This theorem states that the truth-values denoted by these two terms are the same for any assignment of values to the free variables $v_1, \ldots, v_n$. This is usually appropriate for small and relatively simple hardware designs; for more complex designs, it is often impractical to express correctness this way, because in most real products, any

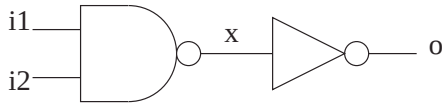


Figure 2: Implementation of two input AND gate

complete logically equivalent specification is likely to be too large and complex to reflect the designer's intent. Hence it is wise to build a partial specification for the design. In this case, the satisfaction relation used to express correctness must therefore express a relationship between a strong constraint (design) and weaker one (specification) rather than strict equivalence. Suppose that $D[v_1, \ldots, v_n]$ and $S[v_1, \ldots, v_n]$ are the design of the device and the partial specification of required behavior respectively. We can formulate this satisfaction relationship as

$$\vdash D[v_1, \ldots, v_n] \Rightarrow S[v_1, \ldots, v_n].$$

2.3 A small example

The basic idea of this approach is to embed both implementation and the specification in higher order logic. The correctness statements, like that every behavior of the implementation satisfies the specification, are then cast in terms of proving some relation in higher order logic. To illustrate this process, we use a trivial example to show many of the underlying ideas.

The task is to show that assuming the NAND gate and the NOT gate behave as specified, then combining them as in Figure 2 yields a two input AND gate. In order to achieve this, we need to carry out four steps:

1. Specify the implementation of the AND gate.
2. Specify the behavioral models for the NAND and NOT gates.
3. Specify the intended behavior of the AND gate.
4. Prove that the implementation satisfies its intended behavior.

There are several ways to specify the implementation of the AND gate in higher order logic. The most common way of doing this is using existential quantification to 'hide' the internal connections, and we would get:

$$\vdash AND\_IMP(i1, i2, o) = \exists x. NAND(i1, i2, x) \wedge NOT(x, o).$$

The behavioral model of the NAND and NOT gates can also be done in many ways in higher order logic. Furthermore, different behavioral models can be used depending on the amounts of details needed or desired. Here, we will use a simple zero-delay model of their behavior. Hence,

Table 2: Proof in higher order logic

Step	Proof	Explanation
0	$AND\_IMP(i1, i2, o)$	assumption
1	$\exists x.NAND(i1, i2, x) \wedge NOT(x, o)$	by def. of AND_IMP
2	$NAND(i1, i2, x) \wedge NOT(x, o)$	strip off $\exists x.$
3	$NAND(i1, i2, x)$	left conjunct of 2
4	$x = \neg(i1 \wedge i2)$	by def. of $NAND$
5	$NOT(x, o)$	right conjunct of 2
6	$o = \neg x$	by def. of NOT
7	$o = \neg(\neg(i1 \wedge i2))$	substitution 4 into 6
8	$o = (i1 \wedge i2)$	simplify using $\neg(\neg(t)) = t$
9	$AND\_SPEC(i1, i2, o)$	by def. of AND_SPEC

$$\vdash NAND(i1, i2, o) = o = \neg(i1 \wedge i2),$$
$$\vdash NOT(i, o) = o = \neg i.$$

In a similar way, the desired behavior of the AND gate can be written as

$$\vdash AND\_SPEC(i1, i2, o) = o = i1 \wedge i2.$$

We are now faced with the task of formally proving that the implementation satisfies the specification. Before we do this, however, we need to define what it means for an implementation to satisfy some specification. Again, there are several ways of expressing this. In this case, we choose to verify that the behavior of the implementation implies the behavior of the specification; thus we want to verify

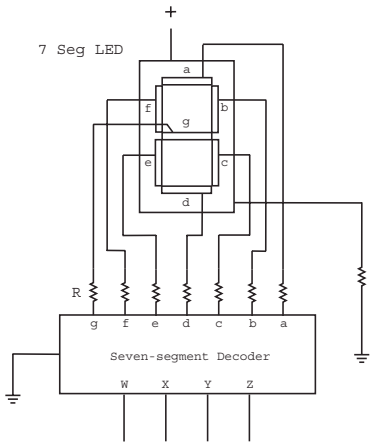
$$AND\_IMP(i1, i2, o) \Rightarrow AND\_SPEC(i1, i2, o)$$

is a valid theorem. A 'hand proof' of this result might look like Table 2.

Although the above manual proof may appear tedious, it is still much shorter than the complete formal proof. The above example is also very simple. However, since we have the full expressive power of higher order logic at our disposal, it is quite simple to generalize the behavioral model for the individual components. In this way, delays and delay models, for example, can be introduced. Of course, the more complex the behavior model is, the more complex the correctness proof will be.[3]

3 The ubiquitous hardware

We illustrate our approach by a case study of the verification of a circuit design of seven-segment LED display decoder [20] [30] as shown below.



A seven-segment LED display is comprised of seven light emitting diodes (LED). Input signals are applied to the input port of the seven-segment decoder, and the decoder translates them into ON/OFF status of the seven LEDs. Then, selected combinations of the LEDs are illuminated to display numeric digits and other symbols.

3.1 From description to specification

The primary function of the decoder is to turn on/off corresponding LEDs based on inputs. Let W, X, Y, Z represent the input port of the decoder, then we get sixteen possible combinations of the four input signals, which means any digit (0 - 9) and some letters (A - F) can be displayed on the seven-segment LED display. Let a, b, c, d, e, f, g represent the output port of the decoder, and let on be 1 and off be 0, then we can create a truth table like Table 3 for describing the intended behavior of the decoder.

3.2 From specification to implementation

Intuitively, the abstraction of seven-segment decoder is a four-input seven-output switching function. One possible approach is to build up the circuit directly from the specification, but here we consider another approach based on partition-and-merge algorithm.[28][5]

First we divide the four-input seven-output switching function into seven four-input one-output normal functions, then implement each function separately. When all functions are ready, we put together all parts and get the final implementation. In this way, the complexity of the design task is greatly reduced. The drawback is probably some redundancy, but this can be refined in the final merging stage.

We shall go into more details of the implementation of one part as an example. The representation function is the abstraction of the intended behavior of LED a, which takes four input signals W, X, Y, Z and generate one output signal a correspondingly.

Table 3: Truth table for the switching function

Display	Input ($W\ X\ Y\ Z$)	Output ($a\ b\ c\ d\ e\ f\ g$)
0	0000	1111110
1	0001	0110000
2	0010	1101101
3	0011	1111001
4	0100	0110011
5	0101	1011011
6	0110	1011111
7	0111	1110000
8	1000	1111111
9	1001	1111011
A	1010	1110111
B	1011	0011111
C	1100	1001110
D	1101	0111101
E	1110	1001111
F	1111	1000111

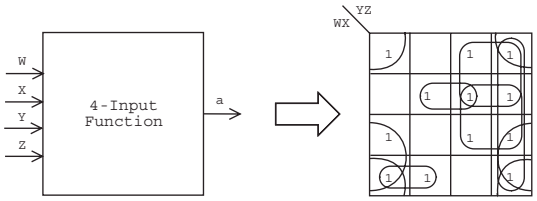


Figure 3: Karnaugh map for 4-input function

1. The first step is to build up the truth table like Table 4 for this four-input one-output function. This is actually a reduced version of Table 3.

2. With the truth table, we can get a logic expression directly.

   $a = \overline{W}\,\overline{X}\,\overline{Y}\,\overline{Z}+\overline{W}\,\overline{X}\,Y\,\overline{Z}+\overline{W}\,\overline{X}\,Y\,Z+\overline{W}\,X\,\overline{Y}\,Z+\overline{W}\,X\,Y\,\overline{Z}+\overline{W}\,X\,Y\,Z+$
   $W\,\overline{X}\,\overline{Y}\,\overline{Z} + W\,\overline{X}\,\overline{Y}\,Z + W\,\overline{X}\,Y\,\overline{Z} + W\,X\,\overline{Y}\,\overline{Z} + W\,X\,Y\,\overline{Z} + W\,X\,Y\,Z$

3. With the initial implementation, we can refine it with emphasis on reducing the sum of minimal terms[13] in order to minimize hardware resource usage. The most common approach is to use a Karnaugh map to achieve this kind of refinement.[2][14] The process is illustrated in Figure 3.

4. After refinement, we get a more concise logic expression:

$$a = Y\,\overline{Z} + \overline{X}\,\overline{Z} + \overline{W}\,Y + X\,Y + W\,\overline{Z} + \overline{W}\,X\,Z + W\,\overline{X}\,\overline{Y}$$

Table 4: Truth table of 4-input function

Input ($WXYZ$)	Output (a)
0000	1
0001	0
0010	1
0011	1
0100	0
0101	1
0110	1
0111	1
1000	1
1001	1
1010	1
1011	0
1100	1
1101	0
1110	1
1111	1

5. Although this refinement result is good enough, we should also consider more practical issues like technology, cost, etc.. Here we choose to make the design mainly with NAND gates.

$$a = \overline{\overline{Y\,\overline{Z}} \cdot \overline{\overline{X}\,\overline{Z}} \cdot \overline{\overline{W}\,Y} \cdot \overline{X\,Y} \cdot \overline{W\,\overline{Z}} \cdot \overline{\overline{W}\,X\,Z} \cdot \overline{W\,\overline{X}\,\overline{Y}}}$$

6. Now it is time to translate the refinement result into schematic design. The diagram is straight forward, as shown in Figure 4.

7. The final step is to design the real circuit based on the schematic design. Here we choose the NAND gate model and the tool discussed in [27][33]

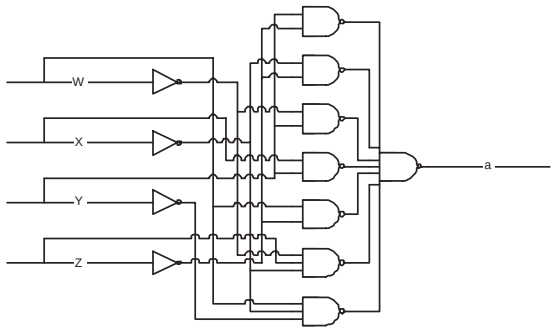


Figure 4: Schematic design for 4-input function

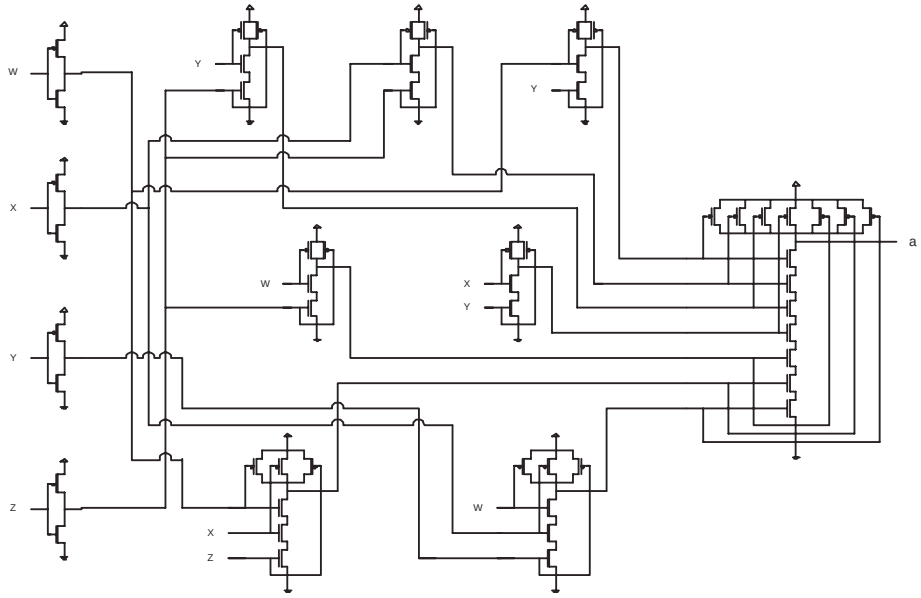


Figure 5: Circuit design for 4-input function

as the atomic element to build up the whole design. The result is shown in Figure 5.

Now the design task of the first part is completed. With the same method, we can design the other six parts:

$$b = \overline{\overline{W\,X} \cdot \overline{X\,Z} \cdot \overline{W\,Y\,Z} \cdot \overline{W\,Y\,Z} \cdot \overline{W\,\overline{Y}\,Z}}$$

$$c = \overline{\overline{W\,\overline{X}} \cdot \overline{\overline{Y}\,Z} \cdot \overline{\overline{W}\,X} \cdot \overline{\overline{W}\,\overline{Y}} \cdot \overline{\overline{W}\,Z}}$$

$$d = \overline{\overline{W\,\overline{Y}} \cdot \overline{W\,\overline{X}\,\overline{Z}} \cdot \overline{X\,\overline{Y}\,Z} \cdot \overline{X\,Y\,\overline{Z}} \cdot \overline{\overline{X}\,Y\,Z}}$$

$$e = \overline{\overline{W\,X} \cdot \overline{Y\,\overline{Z}} \cdot \overline{W\,Y} \cdot \overline{\overline{X}\,\overline{Z}}}$$

$$f = \overline{\overline{\overline{Y}\,\overline{Z}} \cdot \overline{W\,\overline{X}} \cdot \overline{W\,Y} \cdot \overline{X\,\overline{Z}} \cdot \overline{\overline{W}\,X\,\overline{Y}}}$$

$$g = \overline{\overline{W\,\overline{X}} \cdot \overline{Y\,\overline{Z}} \cdot \overline{W\,Z} \cdot \overline{\overline{X}\,Y} \cdot \overline{\overline{W}\,X\,\overline{Y}}}$$

We notice that remaining parts are very similar to the first part in logic expressions, which will also lead to very similar system infrastructures. In order to keep the text concise, we don't list down the designs of the other six parts due to the similarity in these design results.

4 Verification in HOL

HOL is a general theorem proving system developed at the University of Cambridge that is based on higher order logic. HOL is not a fully automated theorem prover but is more than simply a proof checker, falling somewhere between these two extremes. HOL has several nice features as a verification environment:

- Several built-in theories, including booleans, individuals, numbers, products, sums, lists, and trees. These theories build on the five axioms that form the basis of higher order logic to derive a large number of theorems that follow from them.

- Rules of inference for higher order logic. These rules contain not only the eight basic rules of inference from higher order logic, but also a large body of derived inference rules that allow proofs to proceed using larger steps. The HOL system has rules that implement the standard introduction and elimination rules for Predicate Calculus as well as specialized rules for rewriting terms.

- A large collection of tactics. Examples of tactics include `REWRITE_TAC` which rewrites a goal according to some previously proven theorem or definition, `GEN_TAC` which removes unnecessary universally quantified variables from the front of terms, and `EQ_TAC` which says that to show two things are equivalent, we should show that they imply each other.

- A proof management system that keeps track of the state of an interactive proof session.

- A metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be aggregated to form new theories for later use. The metalanguage makes the verification system extremely flexible.

4.1 HOL Overview

The logic of the HOL system is built on higher order logic. The core of the system is rather small. It is built on 5 axioms (Table 5) and 8 rules of inference (Table 6).

The HOL theorem prover uses an ASCII approximation (Table 7) to standard logic notation. One of the types that have been declared is the type of terms in the HOL logic. To enter the term which is a conjunction of two boolean variables A and B, just type the following:

```
- Term `A /\ B`;
> val it = ``A /\ B`` : term
```

Table 5: HOL axioms

1. BOOL_CASES_AX
   $\vdash \exists b : bool.(b = \mathrm{T}) \vee (b = \mathrm{F})$
2. IMP_ANTISYM_AX
   $\vdash \exists b_1 b_2.(b_1 \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow b_1) \Rightarrow (b_1 = b_2)$
3. ETA_AX
   $\vdash \exists f : \alpha \rightarrow \beta.(\lambda x.fx) = f$
4. SELECT_AX
   $\vdash \exists P : \alpha \rightarrow bool.Px \Rightarrow P(\epsilon P)$
5. INFINITY_AX
   $\vdash \exists f : ind \rightarrow ind.\mathrm{One\_One}\, f \wedge \neg(\mathrm{Onto}\, f)$

Table 6: HOL core inference rules

1. Assumption Introduction
   ASSUME: $\frac{}{\{P\} \vdash P}$
2. Reflexivity
   REFL: $\frac{}{\vdash N = N}$
3. Beta Conversion
   BETA_CONV: $\frac{}{\vdash (\lambda v.N) M = N[M/v]}$
4. Abstraction
   ABS: $\frac{\Gamma \vdash M = N}{\Gamma \vdash (\lambda v.M) = (\lambda v.N)}$ (v not free in Γ)
5. Type Instantiation
   INST_TYPE: $\frac{\Gamma \vdash P}{\Gamma \vdash P[\sigma_1,...,\sigma_n/\alpha_1,...,\alpha_n]}$
6. Discharging Assumption:
   DISCH: $\frac{\Gamma \vdash P}{\Gamma - \{Q\} \vdash Q \supset P}$
7. Modus Ponens
   MP: $\frac{\Gamma_1 \vdash P \supset Q \quad \Gamma_2 \vdash P}{\Gamma_1 \cup \Gamma_2 \vdash Q}$
8. Substitution
   SUBST: $\frac{\Gamma_1 \vdash N'_1 \quad ... \quad \Gamma_n \vdash N_n = N'_n \quad \Gamma \vdash P}{\Gamma_1 \cup ... \cup \Gamma_n \cup \Gamma \vdash P[N'_1,...,N'_n/N_1,...,N_n]}$

Table 7: HOL notation

HOL Notation	Standard Notation	Meaning
T	$\top$, true	true
F	$\bot$, false	false
t	$\neg t$	not t
$t_1 \setminus / t_2$	$t_1 \vee t_2$	t_1 or t_2
$t_1 / \setminus t_2$	$t_1 \wedge t_2$	t_1 and t_2
$t_1 ==> t_2$	$t_1 \Rightarrow t_2, t_1 \supset t_2$	t_1 implies t_2
$t_1 = t_2$	$t_1 = t_2$	t_1 equals t_2
$t_1 = t_2$	$t_1 \equiv t_2$	t_1 equivalent to t_2
$\setminus x.t$	$\lambda x.t$	lambda function notation
$!x.t$	$\forall x.t$	t holds for all x
$?x.t$	$\exists x.t$	t holds for some x
$?!x.t$	$\exists! x.t$	t holds for precisely one x
$@x.t$	$\epsilon x.t$	an x for which t holds
if t_1 then t_2 else t_3	$t_1 \rightarrow t_2 \mid t_3$	if t_1 then t_2 else t_3
$t_1 > t_2$	$t_1 > t_2$	t_1 is greater than t_2
$t_1 >= t_2$	$t_1 \geq t_2$	t_1 is greater or equal than t_2
$t_1 < t_2$	$t_1 < t_2$	t_1 is less than t_2
$t_1 <= t_2$	$t_1 \leq t_2$	t_1 is less or equal than t_2

Another way to do the same thing is to type the string we want to parse between the special quotation marks `(--‘` and `‘--)`. We can enter the term as follows:

```
- (--‘A /\ B‘--);
> val it = ‘‘A /\ B‘‘ : term
```

Terms in the HOL logic are represented by the ML datatype `term`. The HOL logic is also typed. The term we just entered was a boolean. The types of the HOL logic are represented by another ML datatype called `hol_type`. The function `type_of: term -> hol_type` will tell you the HOL type of a term.

The HOL logic can be conservatively extended with new types and new constants. The simplest way to add a new constant c is to give a definition of the form $c = t$ where t is a closed term (a term without free variables). An extension by constant definition is always a conservative extension, i.e., it is guaranteed not to introduce inconsistencies.

The ML function used to define a new function is `new_definition:(string * term) -> thm`. For example, a tripling operation can be introduced on the natural numbers by evaluating:

```
new_definition("triple_DEF",(--‘tpl = \x. x + x + x‘--));
```

The constant definition facility also allows arguments to be given on the left hand side; we could have written:

```
new_definition("triple_DEF",(--`tpl x = x + x + x`--));
```

This adds the constant `tpl:num->num` to the logic and stores the definition in the current theory file under the name `triple_DEF`. Note that this does not bind the definition to a name in the current environment (actually, it is bound to the name `it`). If we want to bind the definition to the name `triple_DEF`, then we should evaluate:

```
val triple_DEF =
  new_definition("triple_DEF",(--`tpl x = x + x + x`--));
```

Now suppose we have already decided what goal we would like to prove in HOL and started a proving process by typing `set_goal` command. What would be the best strategy to attack the goal? A very general scheme would be the following:[15]

1. Check whether it is possible to prove (or at least simplify) your goal using existing HOL theorems;

2. If not, expand definitions of all (or some) constants in the goal conclusion;

3. Simplify the goal conclusion (by using beta conversion, removing quantifiers, splitting the goal into simpler subgoals, moving a part of the goal conclusion into the goal assumptions, doing boolean case analysis, ...);

4. Check whether it is possible to prove (or at least simplify) the goal conclusion by rewriting it with trivial rewrites (`REWRITE_TAC []`) and/or the goal assumptions (`ASM_REWRITE_TAC []`);

5. If a goal is still not proved, repeat the procedure starting from the step 1.

4.2 Hardware verification using HOL

The hardware verification process in HOL usually has three steps:[11]

1. Describe the specification

2. Describe the implementation

3. Prove that the implementation meets the specification

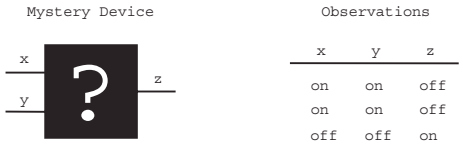


Figure 6: Hardware model in HOL

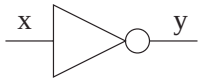


Figure 7: NOT gate

The first step in the verification of hardware is to write a formal specification of the required behavior for the design in HOL. How do we describe a device? The general approach is to model it as a black box in Figure 6. We neglect detailed infrastructure inside the box and only concentrate on its response to the environment outside the box.

Observations of this mystery device can help us to describe hardware in HOL logic:

- Wires can have the value on or off. We model them with boolean variables.
- Devices constrain the values that can be observed on the attached wires. We model these with predicates on wires.

Following this approach, it is possible to express any combinatorial circuit with NOT (Figure 7), AND (Figure 8) and OR (Figure 9) gates, as well as with some means for a line to be tied HI or LO (Figure 10).

```
val NOT_DEF =
  new_definition("NOT_DEF",(--`NOT x x' =
  (x' = ~x)`--));
val AND_DEF =
  new_definition("AND_DEF",(--`AND (x,y) x' =
  (x' = (x /\ y))`--));
val OR_DEF  =
  new_definition("OR_DEF",(--`OR (x,y) x' =
  (x' = (x \/ y))`--));
```

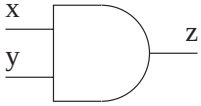


Figure 8: AND gate

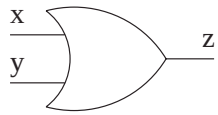


Figure 9: OR gate

Figure 10: Power and ground

```
val HI_DEF  =
  new_definition("HI_DEF",(--‘HI x = (x = T)‘--));
val LO_DEF  =
  new_definition("LO_DEF",(--‘LO x = (x = F)‘--));
```

In practice, it is possible to construct any combinatorial circuit purely from NAND (Figure 11) gates or purely from NOR (Figure 12) gates. And, since it is easier to fabricate circuits that only use one kind of gates, this is what is actually done in industrial practice.

```
val NAND_DEF =
  new_definition("NAND_DEF",(--‘NAND (x,y) x’ =
  (x’ = ~(x /\ y))‘--));
val NOR_DEF  =
  new_definition("NOR_DEF",(--‘NOR (x,y) x’ =
  (x’ = ~(x \/ y))‘--));
```

For example, the implementation of a OR gate by using only NAND gates (Figure 13) can be defined in HOL as below:

```
val OR_IMP = new_definition("OR_IMP",
  (--‘OR_IMP (x, y) x’ = (? a b c d.
    (HI a) /\ (HI b) /\ (NAND (x, a) c) /\
    (NAND (y, b) d) /\ (NAND (c, d) x’))‘--));
```

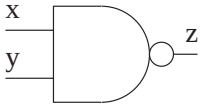


Figure 11: NAND gate

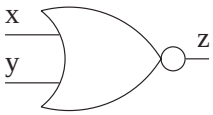


Figure 12: NOR gate

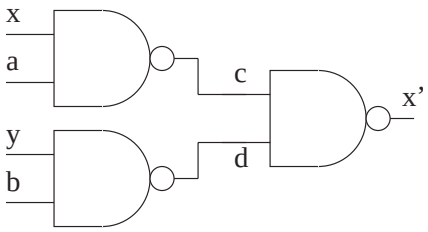


Figure 13: Implementation of OR gate by using only NAND gates

Hereby we can do the verification of the circuit. We would like to know that our design for an OR gate in terms of NAND gates actually functions as an OR gate is supposed to. To establish this fact, we must do the following:

1. Begin the proof by rewriting with definitions.

```
- set_goal([], (--`!x y x'. OR_IMP (x, y) x' ==>
  OR (x, y) x'`--));
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !x y x'. OR_IMP (x,y) x' ==> OR (x,y) x'

     : proofs
- e(REWRITE_TAC[OR_IMP, OR_DEF]);
OK..
1 subgoal:
> val it =
    !x y x'.
      (?a b c d.
       HI a /\ HI b /\ NAND (x,a) c /\
       NAND (y,b) d /\ NAND (c,d) x') ==>
       (x' = x \/ y)

     : goalstack
- e(REWRITE_TAC[HI_DEF, NAND_DEF]);
OK..
```

```
1 subgoal:
> val it =
    !x y x'.
      (?a b c d.
         a /\ b /\ (c = ~(x /\ a)) /\ (d = ~(y /\ b)) /\
        (x' = ~(c /\ d))) ==> (x' = x \/ y)

     : goalstack
```

2. The next step is to strip the goal down to its simplest form.

```
- e(REPEAT STRIP_TAC);
OK..
1 subgoal:
> val it =
    x' = x \/ y
    ------------------------------------
      0.  a
      1.  b
      2.  c = ~(x /\ a)
      3.  d = ~(y /\ b)
      4.  x' = ~(c /\ d)
     : goalstack
```

3. To prove the goal, we may need to use De Morgans Law. [1]

```
- e(ASM_REWRITE_TAC[DE_MORGAN_THM]);
OK..

Goal proved.
 [.....] |- x' = x \/ y

Goal proved.
|- !x y x'.
     (?a b c d.
      a /\ b /\ (c = ~(x /\ a)) /\ (d = ~(y /\
      b)) /\ (x' = ~(c /\ d))) ==> (x' = x \/ y)

Goal proved.
|- !x y x'.
     (?a b c d.
```

[1] Another approach is to use boolean cases analysis. This is the theorem proving equivalent of using truth tables. The tactic is called BOOL_CASES_TAC.

```
          HI a /\ HI b /\ NAND (x,a) c /\
          NAND (y,b) d /\ NAND (c,d) x') ==>
          (x' = x \/ y)
    > val it =
        Initial goal proved.
        |- !x y x'. OR_IMP (x,y) x' ==>
        OR (x,y) x' : goalstack
```

4.3 LED case study

In order to make the verification process simpler, we use a step-wise approach to the whole case. First we prove that the schematic design (Figure 4) satisfies our original description (Table 4). Then we prove that the circuit design (Figure 5) meets the requirements of the schematic design.

The specification of each component and thus the whole schematic design is shown below:

```
val NOT_DEF =
  new_definition("NOT_DEF",
    (--`NOT a x = (x = ~a)`--));
val NAND_DEF =
  new_definition("NAND_DEF",
    (--`NAND a b x = (x = ~(a /\ b))`--));
val NAND3_DEF =
  new_definition("NAND3_DEF",
    (--`NAND3 a b c x = (x = ~(a /\ b /\ c))`--));
val NAND7_DEF =
  new_definition("NAND7_DEF",
    (--`NAND7 a b c d e f g x =
      (x = ~(a /\ b /\ c /\ d /\ e /\ f /\ g))`--));

val LED_A_DEF =
  new_definition("LED_A_DEF",
    (--`LED_A_DEF w x y z a =
      (a = if ((w = F) /\ (x = F) /\ (y = F) /\
      (z = T)) \/
      ((w = F) /\ (x = T) /\ (y = F) /\ (z = F)) \/
      ((w = T) /\ (x = F) /\ (y = T) /\ (z = T)) \/
      ((w = T) /\ (x = T) /\ (y = F) /\ (z = T))
      then F else T)`--));

val LED_A_IMP =
  new_definition("LED_A_IMP",
    (--`LED_A_IMP w x y z a =
```

```
 ?tw tx ty tz t1 t2 t3 t4 t5 t6 t7.
 (NOT w tw) /\ (NOT x tx) /\ (NOT y ty) /\
 (NOT z tz) /\ (NAND y tz t1) /\ (NAND tx tz t2) /\
 (NAND tw y t3) /\ (NAND x y t4) /\ (NAND w tz t5)
  /\ (NAND3 tw x z t6) /\ (NAND3 w tx ty t7) /\
 (NAND7 t1 t2 t3 t4 t5 t6 t7 a)‘--));
```

To facilitate proving process, we try to use several high-level automation tools in the HOL system which allow us to automatically prove or substantially simplify some logical formulas. The most popular automation tools are Simplifier (`simpLib`), Decision Procedures (`decisionLib`), and First-order Prover (`mesonLib`). These three libraries together with some other helpful functions are incorporated into one big library - `bossLib`. With the help of high-level automation tools, the proof length is greatly reduced.

```
- load "bossLib";
- load "simpLib";
- load "mesonLib";
- open bossLib;
- open simpLib;
- open mesonLib;
```

Hereby we can carry out the verification process:

1. Begin the proof by rewriting with definitions.

```
- set_goal([],(--‘!w x y z a.
  LED_A_IMP w x y z a ==> LED_A_DEF w x y z a‘--));
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !w x y z a. LED_A_IMP w x y z a ==>
          LED_A_DEF w x y z a

     : proofs
- e(REWRITE_TAC[LED_A_IMP, LED_A_DEF]);
OK..
1 subgoal:
> val it =
    !w x y z a.
      (?tw tx ty tz t1 t2 t3 t4 t5 t6 t7.
         NOT w tw /\ NOT x tx /\ NOT y ty /\
         NOT z tz /\ NAND y tz t1 /\
         NAND tx tz t2 /\ NAND tw y t3 /\
```

```
          NAND x y t4 /\ NAND w tz t5 /\
          NAND3 tw x z t6 /\ NAND3 w tx ty t7 /\
          NAND7 t1 t2 t3 t4 t5 t6 t7 a) ==>
        (a =
         (if
            ~w /\ ~x /\ ~y /\ z \/ ~w /\ x /\ ~y
            /\ ~z \/ w /\ ~x /\ y /\ z \/ w /\ x
            /\ ~y /\ z
          then
            F
          else
            T))

      : goalstack
- e(REWRITE_TAC[NOT_DEF, NAND_DEF,
    NAND3_DEF, NAND7_DEF]);
OK..
1 subgoal:
> val it =
    !w x y z a.
      (?tw tx ty tz t1 t2 t3 t4 t5 t6 t7.
         (tw = ~w) /\ (tx = ~x) /\ (ty = ~y) /\
         (tz = ~z) /\ (t1 = ~(y /\ tz)) /\
         (t2 = ~(tx /\ tz)) /\ (t3 = ~(tw /\ y)) /\
         (t4 = ~(x /\ y)) /\ (t5 = ~(w /\ tz)) /\
         (t6 = ~(tw /\ x /\ z)) /\ (t7 = ~(w /\ tx
          /\ ty)) /\ (a = ~(t1 /\ t2 /\ t3 /\ t4 /\
          t5 /\ t6 /\ t7))) ==>
      (a =
       (if
          ~w /\ ~x /\ ~y /\ z \/ ~w /\ x /\ ~y /\ ~z
          \/ w /\ ~x /\ y /\ z \/ w /\ x /\ ~y /\ z
        then
          F
        else
          T))

      : goalstack
```

2. Use Simplifier to simplify the expression.

```
- e(SIMP_TAC std_ss []);
OK..
1 subgoal:
```

```
> val it =
    !w x y z.
     y /\ ~z \/ ~x /\ ~z \/ ~w /\ y \/ x /\ y \/
     w /\ ~z \/ ~w /\ x /\ z \/ w /\ ~x /\ ~y =
     (w \/ x \/ y \/ ~z) /\ (w \/ ~x \/ y \/ z) /\
     (~w \/ x \/ ~y \/ ~z) /\ (~w \/ ~x \/ y \/ ~z)

     : goalstack
```

3. Remove universally quantified variables from the front of the subgoal.

```
- e(REPEAT GEN_TAC);
OK..
1 subgoal:
> val it =
   y /\ ~z \/ ~x /\ ~z \/ ~w /\ y \/ x /\ y \/
   w /\ ~z \/ ~w /\ x /\ z \/ w /\ ~x /\ ~y =
   (w \/ x \/ y \/ ~z) /\ (w \/ ~x \/ y \/ z) /\
   (~w \/ x \/ ~y \/ ~z) /\ (~w \/ ~x \/ y \/ ~z)

     : goalstack
```

4. Use boolean cases analysis and rewrite the results.

```
- e(BOOL_CASES_TAC(--`w:bool`--) THEN REWRITE_TAC[]);
OK..
2 subgoals:
> val it =
    y /\ ~z \/ ~x /\ ~z \/ y \/ x
    /\ y \/ x /\ z =
    (x \/ y \/ ~z) /\ (~x \/ y \/ z)


    y /\ ~z \/ ~x /\ ~z \/ x /\ y
    \/ ~z \/ ~x /\ ~y =
    (x \/ ~y \/ ~z) /\ (~x \/ y \/ ~z)

     : goalstack
```

5. Use First-order Prover to prove the goal. (Since we get two subgoals now, we should apply this tactic to both of them.)

```
- e(MESON_TAC[]);
OK..
Meson search level: ......

Goal proved.
|- y /\ ~z \/ ~x /\ ~z \/ x /\ y \/
   ~z \/ ~x /\ ~y = (x \/ ~y \/ ~z)
   /\ (~x \/ y \/ ~z)

Remaining subgoals:
> val it =
    y /\ ~z \/ ~x /\ ~z \/ y \/ x /\
    y \/ x /\ z = (x \/ y \/ ~z) /\
    (~x \/ y \/ z)

     : goalstack
- e(MESON_TAC[]);
OK..
Meson search level: ......

Goal proved.
|- y /\ ~z \/ ~x /\ ~z \/ y \/ x /\
   y \/ x /\ z = (x \/ y \/ ~z) /\
   (~x \/ y \/ z)

Goal proved.
|- y /\ ~z \/ ~x /\ ~z \/ ~w /\ y
   \/ x /\ y \/ w /\ ~z \/ ~w /\ x
   /\ z \/ w /\ ~x /\ ~y =
   (w \/ x \/ y \/ ~z) /\ (w \/ ~x
   \/ y \/ z) /\ (~w \/ x \/ ~y \/
   ~z) /\ (~w \/ ~x \/ y \/ ~z)

Goal proved.
|- !w x y z.
     y /\ ~z \/ ~x /\ ~z \/ ~w /\ y \/
     x /\ y \/ w /\ ~z \/ ~w /\ x /\ z
     \/ w /\ ~x /\ ~y =
     (w \/ x \/ y \/ ~z) /\ (w \/ ~x \/
     y \/ z) /\ (~w \/ x \/ ~y \/ ~z)
     /\ (~w \/ ~x \/ y \/ ~z)

Goal proved.
|- !w x y z a.
```

```
      (?tw tx ty tz t1 t2 t3 t4 t5 t6 t7.
        (tw = ~w) /\ (tx = ~x) /\ (ty = ~y)
         /\ (tz = ~z) /\ (t1 = ~(y /\ tz)) /\
         (t2 = ~(tx /\ tz)) /\ (t3 = ~(tw /\ y))
         /\ (t4 = ~(x /\ y)) /\ (t5 = ~(w /\ tz))
         /\ (t6 = ~(tw /\ x /\ z)) /\ (t7 = ~(w
         /\ tx /\ ty)) /\ (a = ~(t1 /\ t2 /\ t3 /\
         t4 /\ t5 /\ t6 /\ t7))) ==>
      (a =
       (if
          ~w /\ ~x /\ ~y /\ z \/ ~w /\ x /\ ~y
          /\ ~z \/ w /\ ~x /\ y /\ z \/ w /\ x
          /\ ~y /\ z
        then
          F
        else
          T))

Goal proved.
|- !w x y z a.
      (?tw tx ty tz t1 t2 t3 t4 t5 t6 t7.
         NOT w tw /\ NOT x tx /\ NOT y ty /\
         NOT z tz /\ NAND y tz t1 /\
         NAND tx tz t2 /\ NAND tw y t3 /\
         NAND x y t4 /\ NAND w tz t5 /\
         NAND3 tw x z t6 /\ NAND3 w tx ty t7 /\
         NAND7 t1 t2 t3 t4 t5 t6 t7 a) ==>
      (a =
       (if
          ~w /\ ~x /\ ~y /\ z \/ ~w /\ x /\ ~y
          /\ ~z \/ w /\ ~x /\ y /\ z \/ w /\ x
          /\ ~y /\ z
        then
          F
        else
          T))
> val it =
    Initial goal proved.
    |- !w x y z a. LED_A_IMP w x y z a ==>
    LED_A_DEF w x y z a : goalstack
```

The next step is to prove that our circuit design meets all the requirements of our schematic design, where the whole proof is very similar to the above proof. In order to keep the text concise, we don't list down those proofs.

When the verification task of the first part is completed, we verify the other six parts with the same method. For the same reason, here we don't list down the proofs of the other six parts due to the similarity in these verification processes.

5 Verification in PVS

PVS stands for Prototype Verification System, and as the name suggests, it is a prototype environment for specification and verification. The primary purpose of PVS is to provide formal support for conceptualization and debugging in the early stages of the lifecycle of the design of a hardware or software system. The primary emphasis in the PVS proof checker is on supporting the construction of readable proofs[24]. There are some nice features of PVS which make it a popular verification tool:[25]

- An expressive specification language that augments classical higher order logic with a sophisticated type system containing predicate subtypes, and with parameterized theories and a mechanism for defining abstract datatypes such as lists and trees.

- A powerful interactive theorem prover. The basic deductive steps in PVS are large compared with many other systems: there are atomic commands for induction, quantifier reasoning, automatic conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification using binary decision diagrams. Model checking capabilities used for automatically verifying temporal properties of finite state systems are also integrated into PVS.

- A friendly (but not advanced) user interface which is strongly integrated with Emacs.

5.1 PVS overview

The PVS specification language is built on classical typed higher-order logic with the usual base types `bool, nat, rational, real` among others and the function type constructor `[A -> B]`. A distinctive feature of the PVS specification language is predicate subtyping. A subtype {`x: A | P(x)`} consists of exactly those elements `a` of type `A` satisfying predicate `P(a)`. Predicate subtypes are used to explicitly constrain the domain and ranges of operations in a specification and to define partial functions.

A PVS specification consists of a number of theories. A theory is a collection of declarations: types, constants (including functions), axioms that express properties about the constants, and theorems and lemmas to be proved. Theories may import other theories and may be parametric in types and constants.

A proof goal in PVS is represented by a sequent. PVS differs from most proof checkers in providing primitive inference rules that are quite powerful, which also perform steps such as quantifier instantiation, rewriting, beta-reduction, and boolean simplification. Proofs and partial proofs can be saved, edited, and rerun.

To illustrate the above ideas, we consider a simple example to introduce the PVS system. Suppose the file `sum.pvs`[2] contains:

```
sum: THEORY
 BEGIN

 n: VAR nat

 sum(n): RECURSIVE nat =
  (IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF)
  MEASURE id

 closed_form: THEOREM sum(n) = (n * (n + 1))/2

 END sum
```

This specifies a theory called `sum` in which:[18]

1. The variable `n` is declared to have the (predefined) type `nat`;
2. a function `sum` is defined recursively (the well-foundness of the recursion is explicitly justified by the supplied measure - n in this example);
3. a theorem called `closed_form` is conjectured.

If we run PVS on the file `sum.pvs`, an Emacs window containing its contents will pop up. To prove it[3], we type `META-x prove`. This initiates the parsing and typechecking of the theory containing the conjecture. This takes a few seconds and one is then prompted with `Rule?` for a proof command. Responding to it with `(induct "n")` results in the output:[4]

```
Rule? (induct "n")
Inducting on n on formula 1,
this yields  2 subgoals:
closed_form.1 :

  |-------
{1}   sum(0) = 0 * (0 + 1) / 2
```

[2]This file can be found in `./pvs/Examples` directory.

[3]Alternatively, the official proof given by the PVS team can be found in `./pvs/Examples/sum.prf` file.

[4]Alternatively, the proof can be done fully automatically by responding to it with `(induct-and-simplify "n")`.

As in HOL, the subgoals are stacked and the first one is presented to the user, followed by another prompt for a proof command. This goal is solved using PVS proof command `(grind)`. The subgoal is popped and the remaining goal is presented:

```
Rule? (grind)
sum rewrites sum(0)
  to 0
Trying repeated skolemization, instantiation,
and if-lifting,

This completes the proof of closed_form.1.

closed_form.2 :

  |-------
{1}   FORALL j:
        sum(j) = j * (j + 1) / 2 IMPLIES
         sum(j + 1) = (j + 1) * (j + 1 + 1) / 2
```

This is also solved automatically by PVS proof command `(grind)`.

```
Rule? (grind)
sum rewrites sum(1 + j)
  to 1 + j + sum(j)
Trying repeated skolemization, instantiation,
and if-lifting,

This completes the proof of closed_form.2.

Q.E.D.
```

The theory has now been proved, and typing `META-x spt` shows the proof status of the theory:

```
Proof summary for theory sum
   sum_TCC1......proved - complete    [U]( n/a s)
   sum_TCC2......proved - complete    [U]( n/a s)
   closed_form......proved - complete    [O](0.31 s)
   Theory totals: 3 formulas, 3 attempted,
                  3 succeeded (0.31 s)
```

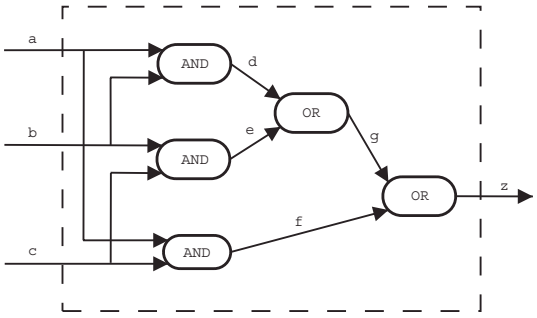


Figure 14: Majority voting circuit[7]

5.2 Hardware verification using PVS

Because the popularity of Gordon's style[19][12] of specifying hardware components in higher order logic, PVS takes the same approach as HOL. The behavior of hardware components is specified by defining predicates that state which combinations of values can appear on their external ports. The behavior of device built by wiring together smaller devices is represented by conjoining the predicates that specify the behaviors of their components with logical conjunction and using existential quantification to hide internal signals.

We illustrate PVS approach by showing a small example of the verification of majority voting circuit[6][7] in PVS. The circuit in Figure 14 is a simplified version of a majority voting circuit as found in nuclear reactors or avionics where three computers each do the same task. If at least two computers signal to do the same thing (i.e. at least two of a, b and c are high) then z is high and the task is performed; otherwise z is low and the task is not performed.[26][23]

We first write the *specification* that asserts the right relationships between inputs (a, b and c) and output (z). The specification is written in a way that is free of implementation detail, and we will not describe any AND/OR gates, just the relationship that ought to hold between the inputs and the outputs. We then write the *implementation* in terms of the AND/OR gates. Finally, we must prove that: *implementation* $\Rightarrow$ *specification*.

The *specification* and *implementation* written in the PVS description language are shown below:

```
1 major_vote: THEORY
2
3   BEGIN
4
5   % input and output variables
6   a, b, c, z: VAR bool
7
8   % conversion function
9   cnf(x: bool): int =
```

```
10      (IF x THEN 1 ELSE 0 ENDIF)
11
12    % specify the required behavior
13    spec(a, b, c, z): bool =
14      z = (cnf(a) + cnf(b) + cnf(c) >= 2)
15
16    % define and_gate
17    and_gate(v, w, x: bool): bool =
18      x IFF (v AND w)
19
20    % define or_gate
21    or_gate(v, w, x: bool): bool =
22      x IFF (v OR w)
23
24    % describe the implementation
25    implementation(a, b, c, z): bool =
26      (EXISTS (d, e, f, g: bool):
27        and_gate(a, b, d)
28        AND and_gate(b, c, e)
29        AND and_gate(c, a, f)
30        AND or_gate(d, e, g)
31        AND or_gate(g, f, z))
32
33    % the result of cnf is either 0 or 1
34    sanity_check_1: THEOREM
35      (FORALL (d: bool): cnf(d) = 0 OR cnf(d) = 1)
36
37    implementation_correctness: THEOREM
38      implementation(a, b, c, z) IMPLIES
        spec(a, b, c, z)
39
40    END major_vote
```

At line 6 we define the boolean variables a, b, c and z. Thus, wherever these variables occur free in the sequel, they will have type $bool$.

In order to write a succinct specification for majority voting, we first define the conversion function *cnf* at line 9 by:

$$\textit{cnf: bool} \rightarrow \textit{int}$$

The function takes an argument of type $bool$ and returns a value of type int. At line 9, the *cnf* function is defined as follows:

cnf(x) = (if x then 1 else 0)

The if/then/else operator takes as its first argument a boolean expression, and as its second and third operator, arguments of type INT. It returns a value of type int. With the help of the conversion function, lines 13 and 14 define *specification* as being a boolean expression in the input and output variables as shown.

We now want to see if we can implement the specification with hardware gates which are defined at line 17 and 21. The boolean expression $(v\ OR\ w)$ at line 22 is a well-formed formula of the PVS logic, where v and w are boolean variables. "OR" is the PVS notation for standard logical $v \vee w$; the same "OR" symbol is also used in the theorem at line 35.

The $implementation$ in terms of AND/OR gates is described at line 25. Implementation correctness, i.e. *implementation* $\Rightarrow$ *specification* is stated as a theorem to be proved at line 37. The two theorems at lines 34 and 37 are proved automatically in this case:

```
sanity_check_1 :

  |-------
{1}   (FORALL (d: bool): cnf(d) = 0 OR cnf(d) = 1)

Rule? (grind)
cnf rewrites cnf(d)
  to (IF d THEN 1 ELSE 0 ENDIF)
Trying repeated skolemization, instantiation,
and if-lifting,
Q.E.D.


implementation_correctness :

  |-------
{1}   FORALL (a, b, c, z: bool):
        implementation(a, b, c, z) IMPLIES
        spec(a, b, c, z)

Rule? (grind)
and_gate rewrites and_gate(a, b, d)
  to d IFF (a AND b)
and_gate rewrites and_gate(b, c, e)
  to e IFF (b AND c)
and_gate rewrites and_gate(c, a, f)
  to f IFF (c AND a)
or_gate rewrites or_gate(d, e, g)
  to g IFF (d OR e)
or_gate rewrites or_gate(g, f, z)
```

```
  to z IFF (g OR f)
implementation rewrites
implementation(a, b, c, z)
  to EXISTS (d, e, f, g: bool):
              d IFF (a AND b) AND e IFF (b AND c)
              AND f IFF (c AND a) AND g IFF (d OR e)
              AND z IFF (g OR f)
cnf rewrites cnf(a)
  to (IF a THEN 1 ELSE 0 ENDIF)
cnf rewrites cnf(b)
  to (IF b THEN 1 ELSE 0 ENDIF)
cnf rewrites cnf(c)
  to (IF c THEN 1 ELSE 0 ENDIF)
spec rewrites spec(a, b, c, z)
  to z =
       ((IF a THEN 1 ELSE 0 ENDIF) +
        (IF b THEN 1 ELSE 0 ENDIF) +
         (IF c THEN 1 ELSE 0 ENDIF)
         >= 2)
Trying repeated skolemization, instantiation,
and if-lifting,
Q.E.D.
```

5.3 LED case study

Follow the same approach as 4.3, first we prove that the schematic design (Figure 4) satisfies our original description (Table 4). Then we prove that the circuit design (Figure 5) meets the requirements of the schematic design. Like 4.3, here we only show the first part of the whole verification.

The schematic components and connections are modeled in PVS[10] as below:

```
logic_gates: THEORY

  BEGIN

  % input and output
  W, X, Y, Z, a: VAR bool

  % define not_gate
  not_gate(i, j: bool): bool =
    j = NOT i

  % define 2 input nand_gate
  nand_gate2(i, j, k: bool): bool =
```

```
    k = NOT (i AND j)

  % define 3 input nand_gate
  nand_gate3(i0, i1, i2, j: bool): bool =
    j = NOT (i0 AND i1 AND i2)

  % define 7 input nand_gate
  nand_gate7(i0, i1, i2, i3, i4, i5, i6, j: bool)
    : bool = j = NOT (i0 AND i1 AND i2 AND i3 AND
                      i4 AND i5 AND i6)

  % specification
  spec(W, X, Y, Z, a): bool =
    NOT a = (W = FALSE AND X = FALSE AND Y = FALSE
             AND Z = TRUE) OR (W = FALSE AND X = TRUE
             AND Y = FALSE AND Z = FALSE) OR
            (W = TRUE AND X = FALSE AND Y = TRUE AND
             Z = TRUE) OR (W = TRUE AND X = TRUE AND
             Y = FALSE AND Z = TRUE)

  % implementation
  imp(W, X, Y, Z, a): bool =
    (EXISTS (tw, tx, ty, tz, t1, t2, t3, t4, t5,
     t6, t7: bool):
       not_gate(W, tw) AND not_gate(X, tx) AND
       not_gate(Y, ty) AND not_gate(Z, tz) AND
       nand_gate2(Y, tz, t1) AND nand_gate2(tx,
       tz, t2) AND nand_gate2(tw, Y, t3) AND
       nand_gate2(X, Y, t4) AND nand_gate2(W,
       tz, t5) AND nand_gate3(tw, X, Z, t6) AND
       nand_gate3(W, tx, ty, t7) AND nand_gate7
       (t1, t2, t3, t4, t5, t6, t7, a))

  implementation_correctness: THEOREM
    imp(W, X, Y, Z, a) IMPLIES spec(W, X, Y, Z, a)


  END logic_gates
```

The proof is automatically done with PVS proof command `(grind)`:

```
implementation_correctness :

  |-------
{1}   FORALL (W, X, Y, Z, a: bool):
```

```
        imp(W, X, Y, Z, a) IMPLIES
        spec(W, X, Y, Z, a)

Rule? (grind)
not_gate rewrites not_gate(W, tw)
  to tw = NOT W
not_gate rewrites not_gate(X, tx)
  to tx = NOT X
not_gate rewrites not_gate(Y, ty)
  to ty = NOT Y
not_gate rewrites not_gate(Z, tz)
  to tz = NOT Z
nand_gate2 rewrites nand_gate2(Y, tz, t1)
  to t1 = NOT (Y AND tz)
nand_gate2 rewrites nand_gate2(tx, tz, t2)
  to t2 = NOT (tx AND tz)
nand_gate2 rewrites nand_gate2(tw, Y, t3)
  to t3 = NOT (tw AND Y)
nand_gate2 rewrites nand_gate2(X, Y, t4)
  to t4 = NOT (X AND Y)
nand_gate2 rewrites nand_gate2(W, tz, t5)
  to t5 = NOT (W AND tz)
nand_gate3 rewrites nand_gate3(tw, X, Z, t6)
  to t6 = NOT (tw AND X AND Z)
nand_gate3 rewrites nand_gate3(W, tx, ty, t7)
  to t7 = NOT (W AND tx AND ty)
nand_gate7 rewrites nand_gate7(t1, t2, t3, t4,
  t5, t6, t7, a) to a = NOT (t1 AND t2 AND t3
  AND t4 AND t5 AND t6 AND t7)
imp rewrites imp(W, X, Y, Z, a)
  to a = NOT ( NOT (Y AND NOT Z) AND NOT
        (NOT X AND NOT Z) AND NOT (NOT W AND Y)
         AND NOT (X AND Y) AND NOT (W AND NOT Z)
         AND NOT (NOT W AND X AND Z) AND NOT
        (W AND NOT X AND NOT Y))
spec rewrites spec(W, X, Y, Z, a)
  to NOT a = (NOT W AND NOT X AND NOT Y AND Z)
             OR (NOT W AND X AND NOT Y AND NOT Z)
             OR (W AND NOT X AND Y AND Z) OR
              (W AND X AND NOT Y AND Z)
Trying repeated skolemization, instantiation,
and if-lifting,
Q.E.D.
```

When the verification task of the first part is completed, we can verify the other six parts with the same method. In order to keep the text concise, we don't list down the proofs of the other six parts due to the similarity in these verification processes.

6 A Comparison of HOL and PVS

There is an overwhelming number of different proof tools available(e.g. in [4] one can find references to over 60 proof tools). All have particular applications that they are especially suited for. Since we have used HOL and PVS as the mechanical verification tools in the previous chapters, hereby it is desirable to do a comparative study of the two proof tools, because both are known as powerful proof tools for higher order logic, which have shown their capabilities in non-trivial applications.

Generally, although HOL and PVS are similar to each other and shares a lot of common features, partly because they are all based on higher order logic and for supporting formal methods applications with proof, there are still some differences. In this section we wish to discuss in some detail our own, more personal, experiences with regards to the case study:

- The meta-language of HOL is ML; hence HOL type system is similar to the type system of ML, which form the basis of the higher order logic theory. (see 4.1).

  PVS is written in Lisp and implements classical typed higher order logic with an extension of predicate subtypes (see 5.1). PVS has many built-in types and uses type constructors to build complex types.

- The specification language of HOL is a ML-style one, which uses the ML datatype `term` to represent the HOL logic; theories are created in ML functions by `new_definition` (see 4.1).

  ```
  val NOT_DEF =
    new_definition("NOT_DEF",
      (--`NOT a x = (x = ~a)`--));
  ```

  Take a look into the case study in 4.3, we can see that the specification consists of the hardware components specification, the target hardware device specification composed with above components' specification, (and the correctness relationship to be proved by `set_goal`, which looks like a part of the proof).

  ```
  set_goal([],(--`!w x y z a.
    LED_A_IMP w x y z a ==> LED_A_DEF w x y z a`--));
  ```

The specification language of PVS is rich, containing many different type constructors and predicate subtypes (see 5.1). Unlike HOL, the syntax is more fixed; many language constructs, such as `IF` and `CASES` are built-in to the language. A specification is usually divided in several theories and theories can import other theories. Although from the case study in 5.3, we can find out that the specification is organized similarly to 4.3, there are two obvious differences:

- Variables have to be declared before using (there is no default datatype mechanism for undefined variables).

```
% input and output
W, X, Y, Z, a: VAR bool
```

- The correctness relationship to be proved is within `THEORY`.

```
logic_gates: THEORY

⋮

implementation_correctness: THEOREM
imp(W, X, Y, Z, a) IMPLIES spec(W, X, Y, Z, a)

END logic_gates
```

- HOL supports both forward and backward proving, but it emphasizes on backward proving by supplying many useful tactics for it. A tactic transforms the proof goal into several subgoals (see 4.2). HOL has a large collection of tactics as well as many proving tools. In the process of proving 4.3, we need to load such tools from libraries by `load` before proving because they don't automatically "stand forward" when applicable.

```
load "bossLib";
load "simpLib";
load "mesonLib";
```

A thorough look of HOL libraries beforehand will help us to get familiar with some of powerful proving tools.

PVS has many tools in the core system which can be automatically invoked (see 5.2). We are quite impressed in the process of proving 5.3; such tools are built-in to the system and are ready to use by invoking `grind` etc.

```
implementation_correctness :

|-------
```

```
{1}FORALL (W, X, Y, Z, a: bool):
   imp(W, X, Y, Z, a) IMPLIES spec(W, X, Y, Z, a)

Rule? (grind)
```

Another difference is that after supplying a tactic, the system repeatedly apply it to the current goal until no changes in the current state. A PVS tactic is like a `REPEAT` HOL tactic in this way.

```
e(REPEAT GEN_TAC);
```

- The most famous difference between HOL and PVS is that the former is a LCF-style prover, which has better security, user extensibility and also ways to import and export proofs to other provers.

When comparing HOL and PVS we realized that both tools had their advantages and disadvantages. If we want to built our own ideal proof tool, it should combine the best of both worlds: [8][29][32]

The logic Predicate subtyping gives so much extra expressiveness and protection against semantical errors, that this should be supported.

The specification language The specification language should be readable, expressive and easily extendible. For function application, we have a slight preference for the bracketless syntax of HOL.

The prover The ideal prover has powerful proof commands for classical reasoning and rewriting, including ordered rewriting. A tactic should return a list of possible next states, as this is useful to try all possible instantiations. Also, decision procedures should be available. Preferably, these decision procedures are not built in to the kernel, but written in the tactical language, so that they can not cause soundness problems. The style of the interactive proof commands of PVS is preferred over that of HOL, because this is more intuitive.

System organization To ensure soundness of the proof tool, the system should have a small kernel. The code of the tool should be freely available, so that users can easily extend it for their own purpose and implement bug fixes.

The proof manager and user interface The tool should keep track of the proof trace. Proofs are best represented as trees, because this is more natural, compared to a linear structure. The tree representation also allows easy navigation through the proof, supported by a visual representation of the tree.

7 Concluding remarks and future work

The paper began with an overview on hardware verification methods, with the emphasis on approaches using higher order logic. We selected two popular verification tools, HOL and PVS, and started with some well-understood, but non-trivial examples, then smoothly moved to a practical verification case study of a seven-segment LED display decoder circuit design.

When applying these two tools to our case, we found PVS was easier to use probably because of some "engineering philosophy" in it. However, we also found that PVS was not an open system, which makes it unsuitable for certain kinds of work requiring more flexibilities. Besides, we also found that there were many opportunities for future work in this case study:

- When writing this paper, I found that today the formal verification community suffers from a lack of meaningful and widely distributed examples for evaluating the performance of verification tools. Existing examples in the area of theorem proving are either toyish or trivial. More realistic hardware examples have little documentation and few property specifications. The benefits of a set of examples are many. It will motivate the development of new algorithms. It will also facilitate comparisons across tools and provide case studies of verification methodologies for users.

- In my opinion, it should be possible to simultaneously ensure the secure extensibility of HOL and the usability and power of PVS. One possible hypopaper is to implement a PVS-style proof environment in HOL.

- Both tools lack a user-friendly interface. PVS is strongly integrated with Emacs. The *de facto* interface for HOL is `hol-mode` (also based on Emacs). There are some more advanced user interfaces based on Tcl/Tk, but they only work for particular versions of HOL.

Over the last two decades hardware verification has moved from academic research to a rapidly growing commercial technology.[16] In the past, verification methods have divided into two well-established approaches: theorem proving and model checking.[9] We focus on theorem proving approach in the whole paper. Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model.

In contrast to theorem proving, model checking is completely automatic and fast, sometimes producing an answer in a matter of minutes. The main disadvantage of model checking is the state explosion problem.

Theorem proving can deal directly with infinite state space. It relies on techniques like structural induction to prove over infinite domains, but theorem provers usually require interaction with a human so that the theorem proving process is slow and often error-prone.

One of the most promising directions in hardware verification is combining model checking and theorem proving, ideally to benefit from the advantages of

both approaches. One way is to employ model checking as a decision procedure within a deductive framework, as is partly done in tools such as HOL and PVS; another way is to use deduction to obtain a finite state abstraction of an implementation that can be verified using model checking.

Another promising direction in hardware verification is to make specification methods and tools more user-friendly. Although industry is adopting techniques like model checking and theorem proving to complement the more traditional one of simulation, there are still a lot of problems for industry applications. (i.e. The notations are too obscure, and the tool is too hard to use.) Ideally, people from industry expect to use the formal hardware specification language as simply a means of communicating ideas to others or of documenting their own designs. They would use tools like model and proof checkers with as much ease as they use compilers. Therefore, we should strive to make our notations and tools accessible to non-experts.

References

[1] A.Gupta. *Formal Hardware Verification Methods: A Survey*. Journal of Formal Methods in System Design, vol. 1, pp. 151 - 238, 1992

[2] C.Max. *Bebop to the Boolean Boogie*. LLH Technology Publishing, 1997

[3] C.-J.H.Seger. *An Introduction to Formal Hardware Verification*. Technical Report, University of British Columbia, Computer Science Department, Number TR-92-13, 1992

[4] Database of Existing Mechanized Reasoning Systems: available at http://www-formal.stanford.edu/clt/ARS/systems.html

[5] D.D.Gajski, F.Vahid, S.Narayan, and J.Gong. *Specification and Design of Embedded System*. Prentice Hall, 1994

[6] D.Gries, F.B.Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, 1993

[7] D.Gries, F.B.Schneider. *Equational Propositional Logic*. available at http://www.ariel.cs.yorku.ca/ logicE/

[8] D.Griffioen, M.Huisman. *A Comparison of PVS and Isabelle/HOL*. Theorem Proving in Higher Order Logics: 11th International Conference, vol. 1479, pp. 123 - 142, Springer, 1998

[9] E.Clarke, J.Wing. *Formal Methods: State of the Art and Future Directions*. CMU Computer Science Technical Report, CMU-CS-96-178, 1996

[10] G.C.Gopalakrishnan. *An Overview of Formal Mathematical Reasoning with Applications to Digital System Verification*. available at http://www.cs.utah.edu/formal_verification

[11] J.Grundy. COMP8033: Mechanical Verification Web Site: http://cs.anu.edu.au/student/comp8033/

[12] J.J.Joyce. *More Reasons Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware*. International Workshop on Formal Methods in VLSI Design, 1991.

[13] J.M.Rabaey. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 2002

[14] KarnaughMap v1.2: available at http://www.puz.com/sw/karnaugh/karnaugh_12.htm

[15] L.Laibinis. Mechanical Verification Course Web Site: http://www.abo.fi/ linas.laibinis/MechVer/MechVer.html

[16] M.Gordon. *21 Years of Hardware Verification*. Talk given at the Royal Society, 1998. available at http://www.cl.cam.ac.uk/ mjcg/BDD/facs21-talk.ps.gz

[17] M.Gordon. *HOL: A Machine Oriented Formulation of Higher Order Logic*. Technical Report 68, Computer Laboratory, University of Cambridge, 1985

[18] M.Gordon. *Notes on PVS from a HOL perspective*. available at http://www.cl.cam.ac.uk/users/mjcg/pvs.ps.gz

[19] M.Gordon. *Why higher-order logic is a good formalism for specifying and verifying hardware*. Formal Aspects of VLSI Design, pp. 153 - 177, North-Holland, 1986

[20] MichiganTech Web Site: http://www.ee.mtu.edu/faculty/schulz/lab_courses/EE2301_fall00/pages/week_4_bcd_to_seven_segment.html

[21] M.John, S.Smith. *Application-Specific Integrated Circuits*. Addison-Wesley, 1997

[22] M.Srivas, H.Rue, D.Cyrluk. *Hardware Verification Using PVS*. Formal Hardware Verification - Methods and Systems in Comparison, Lecture Notes in Computer Science, vol. 1287, pp. 156 - 205, Springer, 1997

[23] N.Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1996.

[24] N.Shankar, S.Owre, J.M.Rushby, D.W.J.Stringer-Calvert. *PVS System Guide*. available at http://pvs.csl.sri.com/doc/pvs-system-guide.pdf

[25] N.Shankar, S.Owre, J.M.Rushby, D.W.J.Stringer-Calvert. *PVS Prover Guide*. available at http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf

[26] N.Storey. *Safety Critical Computer Systems*. Addison-Wesley, 1996.

[27] R.J.Baker, H.W.Li, D.Boyce. *CMOS: Circuit Design, Layout, and Simulation*. John Wiley and Sons publishers, 1998

[28] S.S.Skiena *The Algorithm Design Manual*. Springer-Verlag, 1997

[29] S.Tahar, P.Curzon, and J.Lu. *Three Approaches to Hardware Verification: HOL, MDG and VIS Compared*. Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science, vol. 1522, pp. 433 - 450, Springer, 1998

[30] Tokyo Denki University Web Site: http://www.d.dendai.ac.jp/vhdl/decoder.html

[31] T.Melham. *Higher Order Logic And Hardware Verification*. Cambridge University Press, 1993

[32] V.Zammit. *A Comparative Study of Coq and HOL*. Proceedings of the 10th International Workshop on Theorem Proving in Higher Order Logic. vol. 1275, pp. 323 - 337, Springer, 1997

[33] Windows LASI: layout system for Windows. available at http://members.aol.com/lasicad/index.htm

Turku
Centre *for*
Computer
Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku

- Department of Information Technology
- Department of Mathematics

Åbo Akademi University

- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration

- Institute of Information Systems Sciences

ISBN 952-12-1458-9
ISSN 1239-1891