Using Nuprl for the Verification and Synthesis of Hardware [and Discussion]

Miriam Leeser, T. F. Melham, W. A. Hunt and E. M. Clarke

Phil. Trans. R. Soc. Lond. A 1992 **339**, 49-68
doi: 10.1098/rsta.1992.0025

Using Nuprl for the verification and synthesis of hardware

By Miriam Leeser
School of Electrical Engineering, Cornell University, 335 Engineering and Theory Center, Ithaca, New York 14853, U.S.A.

The Nuprl proof development system, based on constructive type theory, has a sophisticated proof editor and user interface which facilitates the development of proofs and specifications. We present our experience using Nuprl for hardware verification and synthesis. We have verified floating point hardware and are extending this work to reasoning about the IEEE floating point specification. In addition we are using Nuprl to reason about software for synthesizing hardware designs at several different levels. We present two efforts in this area. In the first, we prove a system that synthesizes CMOS circuits from boolean equations. The second system, PBS, minimizes large sets of boolean formulae by using the weak division algorithm.

1. Introduction

Using theorem provers to verify hardware has several advantages over *ad hoc* methods. These include an increased confidence in the correctness of the resulting designs, data-independent analysis and more precise specification of behaviour. Among the significant achievements in this area are the proofs of two microprocessors: the Viper chip with the HOL system (Cohn 1988; Gordon 1985*a*), and the FM8501 design with the Boyer–Moore system (Boyer & Moore 1988; Hunt 1986). Despite such successes, the application of theorem proving to hardware verification has been growing slowly. There are several reasons for this: existing tools are difficult to use, specification techniques may be unnatural, practitioners require an expertise in both hardware and mathematical logic, and the process of proving hardware after it has been designed can be very tedious.

In this paper I discuss research into using Nuprl for formally verifying hardware and hardware synthesis algorithms. Nuprl has a sophisticated proof editor and user interface which facilitates the development of proofs and specifications. Display forms allow the user to define symbols so that proofs appear on screen exactly as they would on paper. Constructive type theory, and in particular, dependent types, allows for a natural specification style. To alleviate the tedium of formal verification, we are concentrating our efforts on providing tools which will allow hardware designers to move verification techniques into the design process. For example, we are developing a toolkit with pre-proven floating point hardware components and a formalized definition of the IEEE floating point specification to minimize the amount of proof required for each floating point design. In addition, we are concentrating on verifying hardware synthesis algorithms which are then used repeatedly. The user of such synthesis procedures gets the advantages of formal methods without requiring any special training. The proof is done once, no matter how many times the synthesis tool is used to generate designs. Nuprl is a very powerful system which is well suited to these tasks.

Nuprl (Constable *et al.* 1986) is a tactic-oriented theorem prover developed at Cornell University. It is a descendent of the LCF project at Edinburgh (Gordon *et al.* 1979) and the PLCV system (Constable & O'Donnell 1978). Nuprl is based on a sequent version of Martin-Löf's constructive type theory (Martin-Löf 1982), a rich and expressive type theory. Nuprl has a user interface built on top of X windows which allows the user to walk over and manipulate the proof tree. Nuprl was developed for automated reasoning in many domains and has integers as a built-in type, as well as decision procedures to support arithmetic reasoning. Our experience has shown that these features make Nuprl a good choice for hardware reasoning.

The research presented here is most similar to work done with the HOL theorem prover (Gordon 1985*a*). HOL is a tactic-oriented theorem prover based on higher-order logic that is also a descendent of LCF. HOL differs from Nuprl in several ways. HOL's logic is classical and is based on a much simpler set of types than Nuprl. HOL was developed explicitly for hardware verification; Nuprl was developed as a general purpose reasoning system. While HOL allows the user access only to the main goal or unproved leaves of a proof, Nuprl provides the user with tools for walking over, examining and manipulating the entire proof tree. HOL has been used to specify and verify several large hardware designs including the VIPER processor (Cohn 1988).

Hardware verification work has also been done with the Boyer–Moore theorem prover (Boyer & Moore 1988). The Boyer–Moore approach differs from that taken in Nuprl and HOL in several ways. The logic of the Boyer–Moore prover is first order and quantifier free. This simpler logic is less expressive, and is principally suited for proofs that can be formulated as induction steps. However, a simpler logic has its advantages: in the Boyer–Moore system, a great deal of automation of proofs is possible. The user interacts with the theorem prover by providing intermediate lemmas and hints for the prover which will allow it to complete the proof automatically. The Boyer–Moore system has been used for several hardware designs including a microprocessor (Hunt 1986). Higher-order logic provides a more natural way of specifying hardware since it allows for easier specification of timing behaviour and abstraction between levels than the logic used in the Boyer–Moore prover. This improvement in specification comes at the cost of losing the ability to provide a high degree of automation for proofs.

Others are investigating the use of Martin-Löf's constructive type theory for reasoning about hardware. Hanna *et al.* (1990*b*) are developing a system called Veritas+. Suk (1991) is using Isabelle (Paulson & Nipkow 1990) to define such a logic. Both these systems take a similar approach to that used in Nuprl. We have an advantage in that Nuprl is much more developed than either of these systems, with large tactic libraries, well-developed decision procedures, and a sophisticated user interface incorporating a proof editor and definition facility.

The rest of this paper is organized as follows. First I go into more depth about the Nuprl system and why it is good for hardware applications. Next I present a survey of research done using Nuprl for hardware. This is split into two sections. The first describes research on verifying hardware. The second describes research on verifying synthesis tools for hardware.

2. Nuprl

Nuprl's logic is a descendent of Martin-Löf's constructive type theory (Martin-Löf 1982). Nuprl supports a rich set of built-in types. Primitive types include `integer`, `atom`, and `void`. Type constructors include *list*, *union*, *function*, *product*, *quotient* and *subset*. Integer induction and list induction are built in.

Types are stratified in an unbounded hierarchy of universes. $\mathtt{U}_1$ is the first universe and contains all small types, including integers, lists, pairs, disjoint union, function space, equality (e.g. `a=b in int`), and first-order propositions. $\mathtt{U}_i$ and all elements of $\mathtt{U}_i$ are in $\mathtt{U}_{i+1}$. This concept of universes allows the Nuprl user to quantify over types in a very natural manner. Quantification over types is not supported by HOL.

Nuprl's logic is higher order. The logic is based on a *propositions-as-types* correspondence: a proposition is true if and only if the type associated with that proposition is inhabited. In other words, a proposition `P` is a type whose elements are proofs of `P`.

A proof in Nuprl may be thought of as a tree. Associated with each node of the tree is a sequent and, if the node is not a leaf, a proof rule. A *sequent* is a number of hypotheses and a goal, and can be written in the form:

$$\mathtt{x}_1\mathtt{:}\ \mathtt{H}_1,\ \mathtt{x}_2\mathtt{:}\ \mathtt{H}_2,\ \ldots,\ \mathtt{x}_n\mathtt{:}\ \mathtt{H}_n\mathtt{>>P.}$$

Here the $\mathtt{H}_i$ are hypotheses and `P` is the conclusion; '>>' is the Nuprl equivalent of a turnstile ($\vdash$). A sequent is true if the conclusion follows from the hypotheses. In constructive logic, this means that given members $\mathtt{x}_i$ of the types $\mathtt{H}_i$ we can construct a member of the type `P`.

Nuprl's proofs are developed in a top–down fashion. The root of the tree is the goal to be proved. The user applies inference rules which refine the goal into subgoals by which the truth of the goal may be established. The children of a node are uniquely determined by the sequent and rule of that node. Below is the fragment of a proof tree where the rule `&-Intro` is applied. The rule refines the goal `P&Q` into two subgoals, one for proving `P`, and the other for proving `Q`.

```
 ...

>>P & Q

BY Intro

|->>P

|->>Q.
```

Inference rules in Nuprl may either be primitive rules or tactics written in the ML programming language. Nuprl tactics are similar to HOL tactics: given a sequent, they apply primitive inference rules and other tactics to the proof tree. Nuprl has several powerful tactics, such as `Autotactic`, which take care of many of the details required in a mechanized proof. `Autotactic` is usually able to completely prove subgoals involving type checking and simple kinds of integer arithmetic and propositional reasoning. Nuprl has a large existing set of tactics which fall into several classes (Howe 1988). Some encode Nuprl's basic logical inference rules.

Others, including a backchaining tactic for second-order matching, incorporate heuristics. It is straightforward to combine existing tactics to create new tactics. Nuprl also provides decision procedures; a typical example is `arith` which simplifies many arithmetic equalities and inequalities.

(a) *Rich set of types*

Constructive type theory is a very expressive type theory which was originally developed to formalize mathematics. Nuprl's types, which include built-in integers, lists, and pairs, are well suited for reasoning about hardware.

Reasoning about hardware frequently requires manipulating booleans and bit vectors as well as arithmetic reasoning. We define booleans as a subset of the integers:

```
{x: int | x=0 in int ∨ x=1 in int}
```

The fact that integers are built-in, combined with arithmetic decision procedures, allows for efficient arithmetic reasoning. Using Nuprl's evaluation facility, we can easily do case analysis over finite types such as booleans.

A great deal of expressive power in constructive type theory is due to dependent types. Dependent types allow for natural and general specifications (Hanna *et al.* 1990*a*). Nuprl provides several dependent types as primitives, including dependent function, dependent product, and subset types. Dependent product types are a generalization of cartesian products where the type of the second element of the pair is dependent on the value of the first element. If the pair `<a, b>` has dependent product type `x:A#B`, then `a` is of type `A` and `b` is of type `B[a/x]`, which is read as '`B` with `a` substituted for `x`'.

A dependent function type, written `x:A->B`, is the type of functions from type `A` to type `B`. Occurrences of the variable `x` which is considered to be of type `A`, are bound in type `B`. For example, if `f` is a dependent function with type `x:A->B` and `a` is a term with type `A`, then `f(a)` has type `B[a/x]`. If `a` has subset type `{x:A|P}` where `A` is a type and `P` is a proposition possibly with a free variable `x`, then `x` has type `A` and `P[a/x]` is a true proposition.

Hanna *et al.* (1990*b*) illustrate the use of dependent types with a definition of the `mod` function for division of non-negative integers. Without dependent types, `mod` would have type `N->N+->N` in Nuprl, where `N` is the type of non-negative integers, and `N+` is the type of positive integers. However, since we know that the result of `mod` is less than its second input, we may wish to encode this in the type. We can express `mod` using the dependent function and subset types:

```
N->m: N+->{n: N | n <m}.
```

We present a simple example for reasoning about busses which illustrates the use of dependent types. This example shows how bounds can be cleanly represented by making the bound a part of the type, as was done in the example above.

We define `int_seg` as a subrange of the integers. This type is defined as the integers between `min` and `max`. `int_seg` is an example of a subset type.

```
int_seg(min,max)=λmin max. {i:int| min ≤i ≤max}.
```

Nuprl has a powerful definition facility which allows the user to define display forms; such features of the user interface are described in more detail below. In this discussion about busses, we use the display form `{min..max}` to represent the integer segment from `min` to `max`. This is the display form defined and used from within Nuprl.

Using `int_seg`, we can define the type `array` which takes a type `A` in U_1 and an integer segment and returns a type; `array`, an example of a dependent function type, is defined by the lambda expression:

```
λA min max. {min..max} -> A
```

A bus is defined as an array of width `w` with type `Bool`:

```
bus(w)=λw. array{0..w-1} of Bool
```

The advantage of this representation is that the width of the bus is part of the type. Checking bus widths when doing manipulations now becomes part of type checking, and allows for very natural statements about bus manipulation. For example, we define an operator `cut`, which selects a subrange of bits from a bus. The function `cut` is:

```
cut(w,min,max,bus_in,bus_out)=
 λ w min max bus_in bus_out.
  ∀j:{0..max-min}. bus_in(min+j)=bus_out(j) in Bool
```

The type of `cut` provides information about the bus widths:

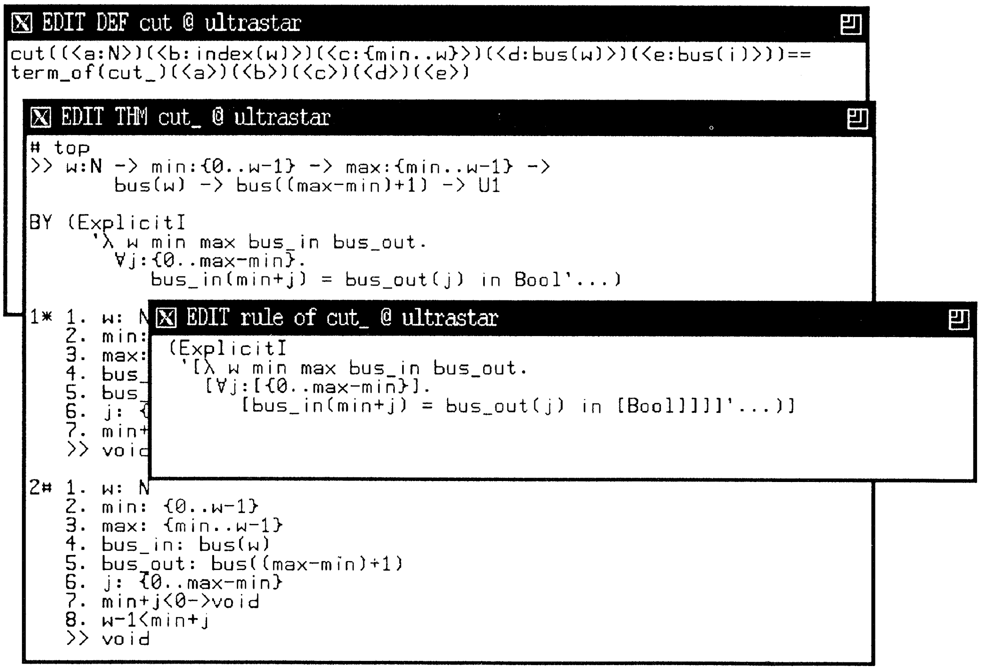
```
w:N -> min:{0..w-1} -> max:{min..w-1} -> bus(w) ->
bus(max-min+1) -> prop
```

We have proved a very straightforward theorem about cut. Namely, that the new bus formed from the cut operation has the correct number and values of wires from the original bus. The goal proved about the cut operator is

```
THM cut_thm
 >>∀w:N.
  ∀min:{0..w-1}.
   ∀max{min..w-1}.
    ∀bus_in:bus(w).
     ∀bus_out:bus(max-min+1).
      cut((w) (min)(max) (bus_in) (bus_out)) =>
       ∀j:{0..max-min}. bus_in(min+j)=bus_out(j) in Bool
```

Using dependent types allows information about a term to be encapsulated in the type of the term. Without dependent types, such information, such as bounds checking, would have to be expressed in separate predicates. In the bus example, all the information about the width of the bus would require separate predicates, and would make for a much lengthier theorem.

The types available in Nuprl are much richer than those available in most other theorem provers used for hardware verification. In HOL, primitive types are *Bool* and *ind*. *num*, *arith* and *list* are theories built on top of the primitive types. HOL provides cartesian product and function type constructors. However, dependent



```
cut((<a:N>)(<b:index(w)>)(<c:{min..w}>)(<d:bus(w)>)(<e:bus(i)>))==
term_of(cut_)(<a>)(<b>)(<c>)(<d>)(<e>)
```

```
# top
>> w:N -> min:{0..w-1} -> max:{min..w-1} ->
       bus(w) -> bus((max-min)+1) -> U1

BY (ExplicitI
      'λ w min max bus_in bus_out.
        ∀j:{0..max-min}.
           bus_in(min+j) = bus_out(j) in Bool'...)

1* 1. w: N
   2. min:
   3. max:
   4. bus_
   5. bus_
   6. j: {
   7. min+
   >> void

2# 1. w: N
   2. min: {0..w-1}
   3. max: {min..w-1}
   4. bus_in: bus(w)
   5. bus_out: bus((max-min)+1)
   6. j: {0..max-min}
   7. min+j<0->void
   8. w-1<min+j
   >> void
```

```
(ExplicitI
  '[λ w min max bus_in bus_out.
    [∀j:[{0..max-min}].
       [bus_in(min+j) = bus_out(j) in [Bool]]]]'...)]
```

Figure 1. A Nuprl session.

types are difficult to express in a logic such as that used in HOL. In Nuprl, the user can quantify over types; such quantification is unavailable in HOL. This richer type theory does not come for free, however. In a simple type theory such as that used in HOL, type checking is decidable. The price for dependent types is that, in general, type checking is undecidable. In practice, type checking can usually be handled automatically by tactics available in Nuprl.

(b) *User interface*

The user interface to Nuprl is built on top of the X window system. Interactions with the system take place within an ordered collection of tactics, theorems and definitions called a library. Objects are created and modified using special purpose structure editors for text editing and proof editing. Figure 1 shows part of the screen during a Nuprl session where the `cut` theorem is being developed.

Nuprl has a sophisticated definition facility for display forms which can greatly increase the readability of complex expressions and hide detail. This definition facility is essentially a macro facility which interacts with the text editor. In the example above we defined the display form of an integer segment from `min` to `max` to be {`min..max`}. This display form is what appears in definitions, theorems and proofs that make use of `int_seg`. Similarly the symbols λ, ∃, ∀ are what appear on the screen when formulae are manipulated. This definition facility allows a user to make the proofs developed in Nuprl look much as they would if the user had done the mathematics on paper.

Nuprl maintains a complete proof tree when proofs are developed. A special purpose proof editor permits the user to browse the proof tree and run tactics at arbitrary nodes. This is in contrast to HOL and most other tactic-oriented theorem provers which maintain just the main goal and the unproven subgoals of a proof. Nuprl provides commands for walking up and down the proof tree, jumping to the top of the tree and jumping to the next unproven node. In Nuprl the user may have several proofs active at once. Transformation tactics are provided for transforming one proof into another. These allow the user to copy pieces of a proof tree and graft them back onto the same or different nodes. These tactics are useful when the user wishes to try out a new tactic on a node without losing any previous work done on the proof. They are also useful when the same tactics are applied to two different branches of the proof tree. This frequently arises when doing case analysis. Such manipulations of the proof tree are unavailable in any of the other theorem provers used for hardware reasoning.

Nuprl tactics have also been developed for outputting LaTeX versions of libraries and proofs. All Nuprl examples in this paper were automatically generated using these tactics.

3. Experience with Nuprl in verification

In this section we describe experience with verification of hardware designs in Nuprl. Our approach to hardware verification is based on that developed by the HOL group at Cambridge (Camillieri *et al.* 1986). Hardware components are modelled as relations on their inputs and outputs, and are either considered primitive or are built from other, simpler hardware components. The specified behaviour of primitive components is assumed to be correct.

For more complex components, the user provides both a specification of intended behaviour and an implementation. The implementation is described as the logical conjunction of simpler components along with behaviours of those components. Internal connections are existentially quantified. This style of specification is hierarchical; the specification of a component at one level of verification becomes part of the implementation at the next higher level. In addition, the user can decide what level of behaviour is primitive. Primitive components may be transistors in one proof, gates in another, register transfer level components in another, etc.

To verify that a component is correct, the user must show that the implementation correctly implements the specified behaviour. In most cases the verification condition we use is that the implementation *implies* the specification. This allows the specification to be more abstract than the implementation.

This approach is for verifying hardware *after* it has been designed. In the remainder of this section we introduce the approach with Nuprl on some simple examples, then describe our experience with some more complex combinational and sequential examples.

As many researchers in the field have noted, such *post hoc* proof can be very tedious and time consuming. In addition, it generally requires a practitioner with training in both hardware design and formal methods; such specialists are rate. Formal methods will become more accepted if we can move them into the design process. This can be accomplished by making specifications as abstract as possible, and by concentrating on proof of components and specifications that can be reused. We are doing this within the realm of floating point design. The final part of this section describes work in progress on a floating point toolkit.

(a) *Using Nuprl for hardware verification*

To specify a hardware component in Nuprl, the user must provide three things: a display form definition, a typing lemma, and an extraction. The typing lemma gives the type of the relation specifying the hardware component. The extraction provides evidence that this type is inhabited. We use the convention that the extraction for a primitive component is a lambda expression specifying its behaviour, and the extraction for a more complex component is a lambda expression describing its structure.

The Nuprl user works within a library. A hardware component is specified as two library objects: a definition, which describes the display form, and a theorem, which is comprised of the type and extraction. We use the convention that the definition has the same name as the component, and the theorem has the name of the component followed by '_'. The user can automatically create these library objects by using the ML function `create_hardware` which takes the component's name, parameters and their types. The ML function creates the definition and sets up the theorem with the correct goal. As a separate step, the user provides the extraction to prove the typing theorem.

To illustrate these concepts we present a simple example, for which the primitive component is an inverter. The user invokes the ML function `create_hardware` with the name `(inverter)` and the parameters and their types `(<a:Bool> -> <y:Bool>)`. The ML function creates the definition `inverter` and the typing theorem `inverter_`. The definition is

```
DEF inverter

  inverter(<a:Bool>,<y:Bool>) ==
  term_of(inverter_)(<a>)(<y>).
```

The expression on the left-hand side of the == symbol is the display form. It is what the user sees or instantiates when using this definition. For hardware components this always has the same form: the name of the module followed by its parameters. The expression on the right-hand side is what this term is defined as. For hardware definitions, a component is defined as the extraction of the typing theorem the user defines. In this definition, `term_of` refers to the extraction of the typing theorem `inverter_`.

The ML function also sets up the typing theorem `inverter_`. For the inverter, the type is a relation over booleans; the type of the relation is `prop`, the type for propositions. The user must open the theorem `inverter_` and provide its extraction. The extraction is evidence that the type `inverter_` is inhabited. Since the inverter is considered primitive, the extraction is its specified behaviour. The complete theorem for the inverter is

```
THM inverter_

   >>Bool -> Bool -> prop

   Extraction:

   λ a y. y=¬(a) in Bool.
```

To specify a buffer built from two inverters, the user provides the name `buffer` and its parameters and their types. The user also specifies the extraction term for the

```
>> ∀a,y:Bool.  buffer((a)(y)) => y=a in Bool
BY (Unfolds [‘buffer‘ ; ‘inverter‘] ...)
|
| 1.   a:Bool
| 2.   y:Bool
| 3.   ∃b:Bool.  b = ¬(a) in Bool & y = ¬(b) in Bool
|->> y = a in Bool
  BY (EThin 3...)
  |
  | 3.   a1:Bool
  | 4.   a1=¬(a) in Bool
  | 5.   y=¬(a1) in Bool
  |->> y = a in Bool
    BY (SubstHypInHypAndThin 4 5 ...)
    |
    | 4.   y=¬(¬(a)) in Bool
    |->> y = a in Bool
      BY (RWHyp rw_not_not_convn 4 ...)
```

Figure 2. Buffer proof.

buffer, which by convention is its structure. The resulting entries in the Nuprl library are:

```
DEF buffer
  buffer((<a:Bool>)(<b:Bool>))==
  (term_of(buffer_)(<a>)(<b>))
THM buffer_
  >> Bool -> Bool -> prop
  Extraction:
  λ a y. ∃b:Bool.inverter(a,b) & inverter(b,y).
```

The behaviour of the buffer is specified in a separate theorem:

```
THM buffer_thm
  >>∀a,y:Bool.buffer((a)(y))  => y=a in Bool.
```

The buffer is verified by proving that the implementation implies the specification. The complete proof in Nuprl is shown in figure 2. The top line, beginning with >>, shows the goal to be proved which has been entered by the user. The user also types in all lines beginning with `BY`. The other lines of the proof are generated by Nuprl.

The lines beginning with `BY` are the rules applied to refine the proof. All the rules in this proof are of the form `(T...)`. This is the display form for running `Autotactic` after the tactic `T`. `Autotactic` is usually able to completely prove subgoals involving type checking and simple kinds of integer arithmetic and propositional reasoning. A subgoal proved with `Autotactic` is not displayed. Underneath each rule are the unproven subgoals generated by applying the rule. Above each subgoal is its hypothesis list which is numbered and displayed vertically.

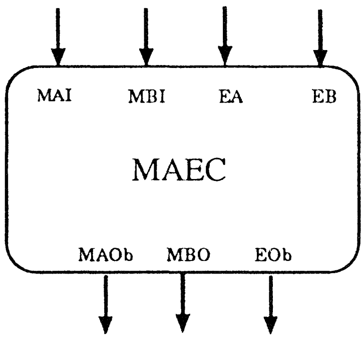


Figure 3. MAEC block diagram.

Numbers in the rules refer to hypothesis numbers. In the interest of brevity, we do not repeat hypotheses which are carried forward from earlier steps in the proof. In general, several subgoals may be generated by each rule, and the proof has the form of a tree. In this proof, only one unproven subgoal is generated at each step. A proof is complete when a rule generates no unproven subgoals.

The proof of the buffer is completed in four steps. In the first step we unroll definitions. In the second step we introduce a witness for the existentially quantified variable (and thin or discard its hypothesis). In the third step we substitute for a variable from one hypothesis to another. The final step involves applying a rewrite rule to simplify the double negation of a Boolean. In general, rewrite rules are based on previously proven lemmas.

(*b*) MAEC *proof*

The previous section illustrated the use of Nuprl for specifying and verifying combinational hardware with a simple example. This same approach has been used to verify much more complex hardware designs. The most complex design verified to date is the MAEC (Mantissa Adjuster and Exponent Calculator) (Basin & DelVecchio 1990). The MAEC is a section of a floating point adder used in a systolic array FFT processor. This processor was developed for NASA as part of an image processing system for ground-based telescopes.

The MAEC inputs the mantissas and exponents for two floating point numbers and adjusts the two mantissas so that the bits with equal weight are aligned in the two mantissas. Each of the mantissas has 49 bits and each exponent nine bits. The MAEC design contains over 5000 transistors. The circuit has 116 inputs and 107 outputs, which makes exhaustive simulation impractical. The Nuprl proof of the circuit goes down to the transistor level, using a simple switch level transistor model.

The block diagram for the MAEC is shown in figure 3. The inputs are the two mantissas, `MAI` and `MBI`, and their respective exponents, `EA` and `EB`. The outputs are `MAOb`, a shifted and inverted copy of `MAI`, and `MBO`, a non-inverted, shifted copy of `MBI`. `EOb` is the inverse of the larger of `EA` and `EB`.

The functions performed by the MAEC are:

(1) it inputs the exponents and mantissas for two floating point numbers;

(2) it right-shifts one of the two mantissas so that bits with equal weight are aligned in the two mantissas;

(3) it outputs (*a*) the shifted mantissa, (*b*) the mantissa that was not shifted, (*c*) the larger of the two exponents.

The description of this behaviour that was verified in Nuprl is shown in figure 4.

```
THM maec_thm
  >>∀ MAI,EA,MBI,EB,MAOb,MBO,EOb:vector.
      maec(MAI,EA,MBI,EB,MAOb,MBO,EOb) =>
        if vec_val(9,EA)≥vec_val(9,EB)
        then ∀ i:{0..8}.  EOb(i)=¬(EA(i)) in Bool &
             ∀ j:{0..48}.  MAOb(j)=¬(MAI(j)) in Bool &
             ∀ k:{0..48}.  MBO(k) = (((k+vec_val(9,EA)-vec_val(9,EB))<49) =>
               MBI(k+vec_val(9,EA)-vec_val(9,EB))|0 ) in Bool
        else ∀ i:{0..8}.  EOb(i)=¬(EB(i)) in Bool &
             ∀ j:{0..48}.  MBO(j)=MBI(j) in Bool &
             ∀ k:{0..48}.  MAOb(k) = (((k+vec_val(9,EB)-vec_val(9,EA))<49) =>
               ¬(MAI(k+vec_val(9,EB)-vec_val(9,EA)))|1 ) in Bool
```

Figure 4. MAEC theorem.

The function vec_val takes a length and a bit vector, and returns its integer value. Its type is N->vector->Int. vec_val is defined as a primitive recursive function with the following behaviour:

```
vec_val(0,vec)  =  0
vec_val(n+1,vec)  =  2ⁿ*bitval(vec(n))  +  vec_val(n,vec),
```

bitval is a function which given a boolean returns its integer value. Since booleans in Nuprl are defined in terms of integers, bitval is simply the identify function.

The MAEC description specifies its behaviour for two cases. The first is the case where exponent EA is greater than or equal to EB, and the second where EB is greater than EA. In the first case, output EOb is an inverted copy of EA, the output mantissa MAOb is an inverted copy of the input mantissa MAI, and the output mantissa MBO is a copy of the input mantissa MBI right shifted by vec_val(9,EA)-vec_val(9,EB) positions. The behaviour of the second case is analogous. When right-shifting is done, any bits shifted to the right of bit position zero are dropped. The leftmost bits are filled with zeros for non-inverted vectors and ones for inverted vectors.

This top level specification formed the basis of the verification of the MAEC, which was performed down to the transistor level. The MAEC proof involves a great deal of bit vector manipulation and arithmetic reasoning.

(c) *Verification of sequential hardware*

The MAEC proof demonstrates that Nuprl can be used for large hardware proofs and for arithmetic reasoning about bit vectors and integers. However, the design is purely combinational. We demonstrate our ability to reason about sequential hardware by outlining the proof of a resettable, sequential counter. An outline of a similar proof done in HOL is available (Gordon 1985*b*). This example illustrates reasoning about circuits with sequential behaviour, vectors and feedback.

For the example we model time with a global clock. Our hardware modules are relations over inputs and outputs which are described as either signals or n-bit wide signals. Signals are functions from times to boolean values; n-bit wide signals (type sig_n) are functions from time to n-bit wide vectors. The types for vector, signal, and sig_n are given below. All are types in U1, the first universe in Nuprl's

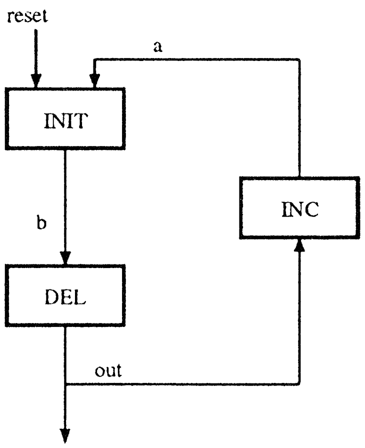


Figure 5. Counter block diagram.

hierarchy of universes. `Bool` is the type of booleans, and `N` is the type of non-negative integers:

```
vector: N -> Bool
signal: N -> Bool
sig_n: N -> vector.
```

The counter outputs a zero on the next clock tick if `reset` is true on the current clock tick. Otherwise it outputs the previous output value plus one modulo $2^{(n+1)}$. The specification of behaviour for the counter in Nuprl is:

```
THM counter_thm
  >>∀n:N. ∀reset:signal. ∀out:sig_n.
    counter((n)(reset)(out))  =>
      ∀t:N.
        if tr(reset(t))
        then vec_val(n,out(t+1))=0
        else vec_val(n,out(t+1))=(vec_val(n,out(t))+1)
        mod 2^(n+1).
```

The type and structure of the counter are described in a theorem named `counter_`.

```
THM counter_
  >> N->signal->sig_n->U1
  Extraction:
  λ n reset out. ∃a,b:sig_n.
    initn_with_t((n)(reset)(a)(b)) &
    deln((b)(out)) &
    inc((n)(out)(a)).
```

The block diagram of the counter implementation is shown in figure 5. The counter is made up of three components, whose types and extractions are given in figure 6.

```
THM initn_with_t_
  >> N->signal->sig_n->sig_n->U1
  Extraction:
  λ n reset a b.  ∀t:N.
    if tr(reset(t))
    then b(t) = zvec(n) in vector
    else b(t)=a(t) in vector

THM deln_
  >> sig_n->sig_n->U1
  Extraction:
  λ a b.  ∀t:N. b(t+1) = a(t) in vector

THM inc_
  >> N->sig_n->sig_n->U1
  Extraction:
  λ n a b.  ∀t:N.
  vec_val(n,b(t))=(vec_val(n,a(t))+1) mod 2^(n+1)
```

Figure 6. Counter component specifications.

`zvec` is a function which given a non-negative integer n, returns a vector of length n whose elements are all zero. For this example, we consider these components to be primitive. We could prove that the behaviours provided are correctly implemented at a lower level of detail. For example, we could verify that the behaviour specified for the `deln` component is correctly implemented by a latch.

The proof of the counter begins by introducing the variables and unfolding the definitions for counter and its components. Next, the proof is split up into two cases; one for reset being true at time t and one for reset being false at time t. The case where reset is true is handled with some straightforward rewriting and `Autotactic`. To prove the other case, we use the following lemma about `zvec`:

```
>>∀n:N. vec_val(n,zvec(n))=0.
```

(*d*) *Toward a toolkit for generating verified floating point designs*

We are aiming our verification efforts at the domain of floating point hardware. This is an especially good domain for theorem proving based verification for several reasons. The designs tend to be datapath oriented and therefore do not lend themselves well to automated verification approaches such as model checking (Clarke *et al.* 1989). Floating point hardware generally requires a large number of inputs and outputs, so exhaustive simulation is impossible. Designers test their floating point hardware by selecting test cases; however, this approach may miss errors. The components used in floating point hardware tend to be drawn from a small set including adders, multipliers, and shifters. This allows for reuse of proof results which is important for speeding up the proof process. Finally, a large amount of the reasoning required is arithmetic, a domain that theorem provers in general are well suited for. Nuprl in particular, with its built in integers and induction schemes, is a powerful theorem prover for arithmetic reasoning.

We have begun formalizing and reasoning about the IEEE floating point specification. Our work on the floating point specification is based on the formalization of the specification in Z (Barrett 1989). This specification was used for the verification of Inmos floating point hardware. This work was a significant success in the application of formal methods to the design process. It differs from our effort in that there was no mechanical support for reasoning about floating point numbers, the translation to hardware was not automated, and the implementation was in microcode. Other research on arithmetic hardware and floating point hardware has been done at the bit manipulation level or the level of reasoning about integers (Pan & Levitt 1990).

Our main contribution thus far has been lifting the reasoning about numerical hardware into the domain of rational numbers. Work at Cornell has demonstrated reasoning about fixed point rational numbers (Jackson 1991) with a case study that illustrates specifying and verifying a floating point multiplier. This work uses dependent types for representing rational numbers.

We are developing a floating point toolkit which will contain pre-proven components such as n-bit wide shifters, multipliers and counters. It will also contain a formally specified version of the IEEE floating point specification with theorems proved about floating point representation and operations. A designer will use the toolkit by connecting the components in the design required, selecting the portions of the floating point specification the design meets, and verifying that the design does in fact meet that specification. We will provide tactics that will be useful for reasoning about floating point hardware. Our approach is intended to be flexible. Users may design with hardware components not available in the library provided that they prove such components. The price of this flexibility is that proofs are not necessarily reused. Similarly, a user may write a specification based on a paradigm other than that of the IEEE standard. The user will then have to formalize their floating point specification, and will not be taking advantage of the specification provided. We believe that many floating point designs will be able to reuse the proofs and specifications provided.

4. Experience with Nuprl in synthesis

Much of the design process involves automated tools for generating designs or pieces of them. Another way to bring verification into the design process is to verify the implementations of these hardware synthesis tools. Others (Hanna *et al.* 1990*b*; Suk 1991; Basin 1991) have discussed using the computational content inherent in constructive proof to derive hardware designs. Nuprl has built-in tools, including an evaluator, to extract and evaluate the computational content derived from a proof. In hardware, this approach appears to work best for module generation, where the designer specifies generic datapaths of arbitrary width. Tactics are used to unroll these generic designs and instantiate the width and the components described. The resulting design depends heavily on the way the design was originally specified and the tactics developed for generating designs. This approach works well for datapaths, but does not generalize to many synthesis tasks in hardware design.

We are investigating a more general approach to verified synthesis. In this section we present two ways we have incorporated synthesis into formal verification. The first is a system written in Nuprl for implementing boolean functions as networks of

```
let BoolSimplify1 = Top (FirstC [AllE;SomeE;Iff;Imp]);;
let NetRealize = LemmaToConv ‘net_realize‘ BoolRel;;
let BoolSimplify2 = Top (FirstC [NotAnd;NotOr;NotNot]);;
let NetExpand = Top (FirstC [PtranAnd;NtranOr;NtranAnd;PtranOr]);;
let NetSimplify = Top (FirstC [CondNtran;CondPtran;NotNot]);;
let Synthesize = BoolSimplify1 THENC NetRealize THENC BoolSimplify2
    THENC NetExpand THENC NetSimplify;;
```

Figure 7. Transistor-level synthesis implementation.

transistors. The second consists of proving an implementation of a conventional synthesis tool using Nuprl. This second approach is likely to have broader applicability to the design task since use of the tool resembles use of a conventional tool and does not require any special training in formal methods.

(*a*) *Synthesizing CMOS circuits from boolean equations*

We have implemented a system in Nuprl that automatically generates transistor-level circuits from boolean expressions such as those output from a logic synthesis system (Basin *et al.* 1991). The translation is driven by the syntax-directed application of synthesis rules. Currently, with only six basic rules, our system generates both series/parallel and pass transistor CMOS networks.

Within Nuprl we have developed a theory of combinational hardware, where the synthesis rules are represented as statements about boolean logic or about the logical equivalence of CMOS representations. The rules are formally proved and are used in a rewrite package which simultaneously applies the transformations and constructs a proof of correctness for each circuit it synthesizes. In addition to guaranteeing that any circuit synthesized correctly implements its logical specification, we also guarantee that the synthesis process always terminates. Our correctness claim is with respect to a switch-level model of transistor behaviour that models boolean behaviour and drive.

The synthesis system is written on top of a rewrite package implemented in Nuprl. The application of the rules for synthesis is a rewrite component that is written in ML and presented in its entirety in figure 7.

Each of the first five lines of ML code in the figure corresponds to one of the five steps of the algorithm. The final line in the program sequences the steps of the algorithm. All of the steps except the second step sequence conversions. Conversions are rewrite functions which are specified by the rules named in the list of arguments. For example, in the step labelled `BoolSimplify2`, the rules applied correspond to DeMorgan's Laws and eliminating double negations. These rules are:

$$\neg(\mathtt{a} \wedge \mathtt{b}) = \neg\mathtt{a} \vee \neg\mathtt{b}$$

$$\neg(\mathtt{a} \vee \mathtt{b}) = \neg\mathtt{a} \wedge \neg\mathtt{b}$$

$$\neg\neg\mathtt{a} = \mathtt{a}.$$

The second step implements the boolean expression as a network of transistors using an ML program which transforms a Nuprl lemma to a conversion.

This implementation is concise and understandable as well as extensible. New boolean optimizations or transformation rules may be added to the system by proving them correct and adding the corresponding transformation to the appropriate rewrite step. With the few rules included so far we have been able to synthesize many optimized circuits including adders and barrel shifters.

(*b*) *PBS: proven boolean simplification*

PBS (Aagaard & Leeser 1991) is a proven implementation of the weak division method for simplifying combinational hardware. Weak division is widely used in logic synthesis packages; most notably in MIS (Brayton *et al.* 1987). Weak division is a good candidate for formal verification because the mathematics is based on the theories of boolean algebra. In addition, logic synthesis is commonly automated and the resulting CAD tools are reused many times. We have used Nuprl to prove the implementation of PBS used in the BEDROC synthesis system (Leeser *et al.* 1991).

Weak division minimizes the area used by combinational logic. It does this by finding common subcircuits in a circuit and implementing the common logic once. This technique results in a great reduction in the area required to implement a design. In PBS, we have proven the implementation of weak division, not just the algorithm. Every time PBS is run, the user has confidence that the output is functionally equivalent to the input and that the output is irredundant. Thus, though a large amount of effort was put in to developing the proof, the results of the proof are used over and over again.

We proved two facts about the PBS implementation. The first is that the set of equations input to PBS is functionally equivalent to the set of equations output by PBS. The second is that the output is minimal in that the largest common divisor of any two boolean functions is a single literal. This can be stated formally as: *the size of the support of the intersection of any two divisors of any two functions in the output system* s *is at most one*. We write this in Nuprl as:

```
minimality_pbs(s:syst_t) =
  ∀f1,f2: fnct_t.
    f1 ∈ s ∧ f2 ∈ s =>
      f1 ≠ f2 =>
        ∀d1, d2: expr_t.
          d1 ∈ δ (f1) ∧d2 ∈ δ(f2) =>
            |sup(d1 ∩ d2)|≤1.
```

Users of logic synthesis tools usually check the functional equivalence of their input and output systems by simulation, or with BDD-based algorithms (Bryant 1986). Such checking needs to be done for every circuit generated; the proof of PBS was done once, and guarantees functional equivalence for all sets of input and output equations. Perhaps more interesting, however, is the minimality result.

Circuits are completely single stuck-at fault testable if and only if they are minimal by the above criteria, which is also called irredundancy (Hachtel *et al.* 1989). In the proof of PBS, it was verified that the outputs of PBS will always be irredundant. We can therefore guarantee that all circuits output by PBS are completely single stuck-at fault testable. It is much easier to verify properties of the implementation, such as irredundancy, using theorem proving than on a case by case basis.

The implementation of PBS is in the Standard ML programming language (SML). The proof consists of embedding a subset of the SML language in Nuprl and verifying that the implementation of PBS done in that SML subset is correct. The

```
mem(eq_fn, a, a_list)=
  let
    fun f eq_fn a hd result =
      eq_fn(a)(hd) orelse result
  in
    reduce (f(eq_fn)(a)) false a_list
  end
```

Figure 8. Definition of membership function.

```
mem_x_nil
  ∀A : U1.
    ∀eq_fn :  A -> A -> Bool.
      ∀a :  A.
        is_eq_rel(A)(eq_fn) =>
          ¬(tr(mem(eq_fn)(a)(nil)))
```

Figure 9. Theorem for membership in an empty list.

```
mem_x_ht
  ∀A:U1.
    ∀eq_fn:A->A->Bool.
      ∀tl:A list.
        ∀hd,a:A.
          is_eq_rel(A) (eq_fn) =>
            tr(mem(eq_fn)(a)(hd::tl)) <=>
            tr(eq_fn(a,hd)) ∨
              tr(mem(eq_fn)(a)(tl))
```

Figure 10. Theorem for membership in a non-empty list.

implementation of weak division is made up of a hierarchy of smaller algorithms. Each function of the SML implementation was copied into Nuprl and then theorems describing the behaviour of the functions were proved. The overall proof proceeded in a bottom-up fashion.

A simple function in the proof of PBS is the membership function for lists (mem), whose definition is shown in figure 8. This function takes three parameters: an equality function (eq_fn), a test element (a) and a list (a_list). The function uses the higher-order function reduce to recurse over the list a_list and test if there is an element of a_list which is equal (according to the equality function eq_fn) to the test element a. The function reduce is used to handle most cases of list recursion in PBS. The definition of reduce is

```
reduce f nil_val nil  = nil_val
reduce f nil_val hd::tl  = f hd (reduce nil_val f tl).
```

There are two principal lemmas used to describe the function mem. The first one, mem_x_nil, states that an empty list does not have any members (figure 9). The second theorem, mem_x_ht (figure 10), states that an element is a member of a non-empty list if and only if it is equal to the head of the list, or it is a member of the tail of the list. The proof of `mem` is typical of the proofs in PBS, and makes use of unrolling of definitions, list induction, rewriting, and `Autotactic`.

5. Conclusions

In this paper we have illustrated the use of Nuprl for hardware verification and synthesis. Nuprl is an extremely powerful theorem prover for these applications. This is due to many factors including its rich type theory, arithmetic decision procedures and user interface.

As tools for theorem proving based verification improve, and more libraries, decision procedures and examples become available, theorem proving will be applied more and more to guaranteeing the correctness of hardware. However, due to the tedium required to use any theorem prover, and the level of sophistication required of the user, these efforts will be applied only to subsets of hardware for which they are particularly well suited. Areas that can make use of proof done in advance, or reuse existing proofs will benefit greatly from this approach. Floating point hardware is one such area with its well-defined specifications, arithmetic reasoning, and small set of hardware components which are reused in many designs. The future of theorem proving based hardware reasoning will be in such areas that can exploit reusable proof.

This work owes a great debt to Mike Gordon and the research done in hardware verification at Cambridge University. This paper describes work done at Cornell by many researchers. The MAEC proof was done by Pete DelVecchio and Dave Basin. The floating point arithmetic reasoning is currently being worked on by Paul Jackson. Dave Basin implemented the transistor level synthesis system in Nuprl. The proof of PBS was done by Mark Aagaard. We would like to thank members of the Nuprl group, especially Robert Constable, Doug Howe, and Chet Murthy for many useful discussions. We also thank Albert Camilleri for useful discussions about the HOL system. Mark Aagaard helped with use of the Nuprl system and developed many of the examples in this paper. Thanks to Mark Aagaard, Robert Constable and Robert Cooper for their comments on earlier drafts of this paper.

References

Aagaard, M. & Leeser, M. 1991 A formally verified system for logic synthesis. In *International Conference on Computer Design*. IEEE, October 1991.

Barrett, G. 1989 Formal methods applied to a floating-point number system. *IEEE Trans. Software Engng* **15**, 611–621.

Basin, D. 1991 Extracting circuits from constructive proofs. In *International Workshop on Formal Methods in VLSI Design*. ACM, 1991.

Basin, D. A. & DelVecchio, P. 1990 Verification of combinational logic in Nuprl. In *Hardware specification, verification, and synthesis: mathematical aspects* (ed. M. E. Leeser & G. M. Brown), pp. 333–357. Springer Verlag, LNCS 408.

Basin, D. A., Brown, G. M. & Leeser, M. E. 1991 Formally verified synthesis of combinational CMOS circuits. *Integration VLSI J.* **11**, 235–250.

Boyer, R. S. & Moore, J. S. 1988 *A Computational logic handbook*, vol. 23 of *Perspectives in computing*. Academic Press.

Brayton, R. K., Rudell, R., Sangiovanni-Vincentelli, A. & Wang, A. R. 1987 MIS: a multiple-level logic optimization system. *IEEE Trans. CAD* **CAD-6**, 6.

Bryant, R. E. 1986 Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers* **C-35**, 677–691.

Camillieri, A. J., Gordon, M. J. C. & Melham, T. F. 1986 Hardware verification using higher-order logic. In *From HDL descriptions to guaranteed correct circuit designs* (ed. D. Borrione) North Holland.

Clarke, E. M., Long, D. E. & McMillan, K. L. 1989 Compositional model checking. In *Fourth IEEE Symposium on Logic in Computer Science.*

Cohn, A. 1988 A proof of correctness of the VIPER microprocessor: the first level. In *VLSI specification, verification, and synthesis*, pp. 27–71. Kluwer Academic.

Constable, R. L. & O'Donnell, M. J. 1978 *A programming logic*. Cambridge, MA: Winthrop.

Constable, R. L. *et al.* 1986 *Implementing mathematics with the Nuprl proof development system.* Prentice Hall.

Gordon, M. J. C. 1985*a* HOL: a machine oriented formulation of higher order logic. *Tech. Rep.* 68. Cambridge University Computer Laboratory.

Gordon, M. J. C. 1985*b* Hardware verification by formal proof. *Tech. Rep.* 74. University of Cambridge Computer Laboratory.

Gordon, M., Milner, R. & Wadsworth, C. 1979 *Edinburgh LCF: a mechanized logic of computation*, vol. 78. Lecture Notes in Computer Science. Springer Verlag.

Hachtel, G., Jacoby, R., Keutzer, K. & Morrison, C. 1989 On the relationship between area optimization and multifault testability of multilevel logic. In *International Conference on Computer Aided Design*, pp. 316–319. ACM/IEEE.

Hanna, F. K., Daeche, N. & Longley, M. 1990*a* Specification and verification using dependent types. *IEEE Trans. Software Engng* **16**, 949–964.

Hanna, F. K., Daeche, N. & Longley, M. 1990*b* Veritas+: a specification language based on type theory. In *Proceedings of the MSI Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, pp. 358–379. Springer Verlag. LNCS 408.

Howe, D. 1988 Automating reasoning in an implementation of constructive type theory. Ph.D. thesis, Cornell University, Ithaca, New York.

Hunt Jr, W. A. 1986 FM8501: a verified microprocessor. Ph.D. thesis, Institute for Computing Science, The University of Texas at Austin.

Jackson, P. 1991 Developing a toolkit for floating-point hardware in the Nuprl proof development system. In *Proceedings of the Advanced Research Workshop on Correct Hardware Design Methodologies.*

Leeser, M., Aagaard, M., Linderman, M., Chapman, R., Johnson, R. & Meier, S. 1991 The BEDROC high level synthesis system. In *ASIC'91. IEEE.*

Martin-Löf, P. 1982 Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy*, pp. 153–175. North-Holland.

Pan, J. & Levitt, K. N. 1990 A formal specification of the IEEE floating-point standard. In *24th Asilomar Conference on Signals, Systems, and Computers.*

Paulson, L. & Nipkow, T. 1990 Isabelle tutorial and user's manual. *Tech. Rep.* University of Cambridge Computer Laboratory.

Suk, D. 1991 Hardware synthesis in constructive type theory. In *Designing correct circuits, Oxford 1990* (ed. G. Jones & M. Sheeran). Springer-Verlag.

Discussion

T. F. Melham (*University of Cambridge, U.K.*). Professor Leeser has shown that one advantage of dependent types is that they can be used to organize the content of theorems, essentially by separating information about structure (for example, bus widths) from information about function. Does she think that this can also be done in a logic without dependent types – for example, simple type theory – by simply adopting appropriate conventions for formulating and organizing theorems?

M. Leeser. I think Dr Melham's question sums up quite nicely what we are doing. In the example, I gave for using dependent types, we used the duality of the type system and the proof logic to separate structural information from functional information. This could be done in a logic without dependent types by carefully structuring theorems. However, the specification style with dependent types is quite natural as a result of this duality. Ease of specification is one of our goals, as well as ease of exploiting the specification. Both are a result of using dependent types. With the bus example, the proof obligations having to do with structure, such as bounds checking, are proof obligations about types. They do not clutter up the proof of functionality.

W. A. Hunt (*Computational Logic, Inc., U.S.A.*). Describe the proof that a minimalized circuit can be completely single stuck-at fault tested (Hachtel *et al.* 1989).

M. Leeser. Briefly, the result states that a circuit which is guaranteed minimal, in the sense that all common divisors contain at most one literal, is completely single stuck-at fault testable. In our proof of PBS, we prove that the results of PBS will always be minimal, therefore we use the result of Hachtel to claim that the circuits will also be testable. This shows one way that formal verification and testing interact. Such interactions should be investigated further.

Additional reference

Hachtel, G., Jacoby, R., Keutzer, K. & Morrison, C. 1989 On the relationship between area optimization and multifault testability of multilevel logic. In *International Conference on Computer Aided Design*, pp. 316–319. ACM/IEEE.

E. M. Clarke (*Carnegie Mellon, U.S.A.*). What benefit is to be gained from using a constructive logic?

M. Leeser. All of the results discussed in this paper could equally have been arrived at by using a classical logic with dependent types, such as that used in the Veritas+ system. Since classical logic is a subset of constructive logic, we do not lose anything by using a constructive logic. As a practical matter, I have not found that using a constructive logic has got in the way at all. So far, we have not taken advantage of the fact that the logic is constructive. It is possible that such benefits could be gained in the future. Basin at the University of Edinburgh and Suk at Chalmers University in Sweden are investigating using the computational content inherent in constructive proof to derive hardware designs. So far, this approach has been applied to a very limited number of circuits. Taking advantage of the constructive content is an interesting area for future research.