More Reasons
Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware

by

Jeffrey J. Joyce

Technical Report 90-35
November, 1990

Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1W5

More Reasons
Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware

Jeffrey J. Joyce

Department of Computer Science
University of British Columbia
Vancouver, B.C., CANADA V6T 1W5

phone: (604) 228-4327
e-mail: joyce@cs.ubc.ca

1 Introduction

More than five years ago, Mike Gordon published a paper on "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware" [15]. At that time, the concept of formal hardware verification was in its infancy and, in particular, the idea of using higher-order logic as a hardware description language (due originally to Keith Hanna [22]) was largely unexplored. The purpose of Gordon's paper was to show that pure logic, both as a specification language and as a deductive system, is completely adequate for specifying and verifying hardware; that specialized description languages and specialized deductive systems are not needed.

Since the publication of Gordon's 1985 paper, the idea of using higher-order logic for specifying and verifying hardware has been explored by a number of researchers. One of the best known (and most impressive) examples of the higher-order logic approach is Avra Cohn's partial verification of the commercially available Viper microprocessor [7, 8, 9]. Higher-order logic has also turned out to be a very good formalism for reasoning about many other kinds of systems besides hardware. This includes reasoning about software [16], process algebras [5, 18], compiling algorithms [26], and communication protocols [6].

The main purpose of this paper is to argue that higher-order logic, compare to less expressive formalisms such as first-order logic, is a very good formalism for specifying and verifying hardware. We focus on two main reasons:

- The ability to support generic specifications of hardware, that is, hardware specifications parameterized (directly or indirectly) by functions and by data types.
- The ability to embed special-purpose formalisms such as temporal logic.

We have experimented with both generic specification and embedded formalisms in the context of specifying and verifying a very simple microprocessor [27]. The arguments presented in this paper are largely based on experience with this particular case study.

2 Higher-Order Logic

Higher-order logic extends first-order logic by allowing variables to range over functions and predicates. Many basic concepts of higher-order logic are inherited from first-order logic. Other concepts, such as the association of types with terms, are familiar ideas from experience with strongly-typed programming languages. Much of the actual notation of higher-order logic, in particular, the formulation described by Gordon for the Cambridge HOL system [17], should be familiar from informal mathematics, e.g., $\forall$, $\exists$, $\wedge$, $\vee$, $\neg$, $\Longrightarrow$.

Allowing variables to range over functions and predicates allows functions and predicates to be quantified. It also allows (higher-order) functions and (higher-order) predicates to be parameterized by other functions and predicates. A (higher-order) function may return another function or predicate as a result.

The particular formulation of higher-order logic used in the HOL system is based on five axioms and eight primitive inference rules. The implementation of the HOL system uses this axiomatization as a basis for guaranteeing *proof security*: every theorem generated by the HOL system is indeed a theorem of higher-order logic. This approach (shared with several other verification systems) may be contrasted with verification systems that are implemented as a (possibly *ad hoc*) collection of decisions procedures.

Although more expressive than many hardware specification languages, the HOL logic is less expressive than the constructive type theory approaches described by Hanna [23], and by Basin, Brown and Leeser[2]. In particular, these approaches allow for *dependent types* which may be useful for some aspects of hardware specification. However, the price of this increase in expressive power is the loss of decidable type-checking.

3 Generic Specification

Generic description is already established as a powerful concept in many high-level programming languages. For instance, the 'generic mechanism' of Ada allows a subprogram or package to be parameterized by types and subprograms as well as values and objects. This feature supports modularity and abstraction and provides a convenient mechanism for the reliable re-use of software.

Elsewhere [27, 28], we have argued that generic description is also powerful concept in the

context of formally verifying digital hardware. In addition to the well known advantages of modularity, abstraction and reliable re-usability, the use of generic specification to eliminate non-essential detail from a formal specification sharpens the distinction between what has and what has not been formally considered. In a hierarchical proof effort, the elimination of non-essential detail isolates each level from details only relevant to other levels. Finally, the elimination of non-essential detail reduces the need for special-purpose proof infrastructure for reasoning about hardware, e.g., hardware-oriented data types.

Most languages used for specifying hardware, including conventional languages such as VHDL[1], do not provide explicit mechanisms for supporting and encouraging fully generic specification. (OBJ [13] and EHDM [37] are among the few specification languages with explicit mechanisms for generic description.) However, we have proposed a technique for expressing genericity in any language with (at least) the expressive power of higher-order logic in a manner that avoids the need for explicit mechanisms to support generic specifications. This technique (described fully in [27, 28]) is based on the use of higher-order predicates parameterized by function variables and type variables. We believe that this technique is a very direct (if not the most direct) way to specify hardware generically.

Unlike ordinary simulation, formal verification does not require every symbol appearing in a formal specification to be fully defined. Often, symbols representing various data types or operations, e.g., arithmetic operations performed by an ALU, are only used as 'place-holders' in a correctness proof. This is particularly true in the case of very large verification problems which are organized into a hierarchy of verification problems. For instance, the formal verification of a microprocessor can be organized into a hierarchy of verification problems corresponding to the conventional view of a microprocessor as a hierarchy of interpreters [1]. In such a hierarchy, many data types and operations are typically used only as place-holders when deriving higher level correctness results. The used of defined symbols as mere place-holders is evident in the formal verification of the major state machine of the Viper microprocessor [8]. Cohn reports that this level of correctness proof was only concerned with the flow of control:

> *"There was no computations of values at the major state level - that is, additions, comparisons, shifts and so on, [...] the proof did not require analysis of the functions the arithmetic-logic unit."*

In a conventional approach to hardware specification, every data type and every operator is fully defined whether or not it needs to be defined. These symbols are fully defined even if they will only serve as place-holders in the correctness proofs. This *closed-world* approach may be partly explained by the influence of "the more detailed, the better" attitude carried over from experience with multi-level simulation [12]. Some may also believe (unaccountably) that the use of fully defined types (even when their definitions are not used) is an extra measure of confidence in the correctness result.

[1] Limitations of VHDL with respect to fully generic hardware description are discussed in [29].

In contrast to this conventional style of specifying hardware, we advocate a generic approach where hardware specifications are parameterized (as much as possible) without changing the essence of the verification problem. For data types and operators that are used merely as place-holders, we advocate treating them literally as place-holders, that is, as *uninterpreted types* and *uninterpreted operations.*

The generic specification of hardware (as well as software) falls naturally within the domain of higher-order logic. Generic specifications are easily created by parameterizing specifications by functions and types. This can be illustrated by the formal specification of a simple ALU (the ALU used in the formal specification of the formally verified TAMARACK-3 microprocessor described in [27]).

We first show how this ALU can be specified in a conventional style where every data type and every operator is fully defined. We assume that the functions `INC16`, `ADD16` and `SUB16` have been previously defined as arithmetic operations on a previously defined type `:word16` (a data type representing 16-bit words). We also assume that `ZERO16` is a previously defined constant denoting the representation of zero as a 16-bit word.

```
Define (
  "ALU (f0,f1,inp1,inp2,out) =
    ∀t:time.
      out t =
        (((f0 t,f1 t) = (T,T)) ⇒ (INC16 (inp2 t)) |
         ((f0 t,f1 t) = (T,F)) ⇒ (ADD16 (inp1 t,inp2 t)) |
         ((f0 t,f1 t) = (F,T)) ⇒ (SUB16 (inp1 t,inp2 t)) |
                                  ZERO16")
```

The above specification can be transformed into a generic specification by replacing `INC16`, `ADD16`, `SUB16` and `ZERO16` with the function variables `incfun`, `addfun`, `subfun` and `zerofun` (where `zerofun` is a 0-place function) to represent the arithmetic operations performed by this ALU. This generic specification does not specify any details about these operations. Furthermore, instead of a defined data type, the generic specification is parameterized (implicitly) by the type variable `:*word`.

```
Define (
  "ALU (incfun,addfun,subfun,zerofun) (f0,f1,inp1,inp2,out) =
    ∀t:time.
      out t =
        (((f0 t,f1 t) = (T,T)) ⇒ (incfun (inp2 t)) |
         ((f0 t,f1 t) = (T,F)) ⇒ (addfun (inp1 t,inp2 t)) |
         ((f0 t,f1 t) = (F,T)) ⇒ (subfun (inp1 t,inp2 t)) |
                                  zerofun")
```

Type variables and function variables are used in the above specification to stand for *uninterpreted types* and *uninterpreted operators*. The transformation of a conventional specification of this simple ALU into a generic specification only requires the addition of a few extra parameters to the parameter list of the predicate `ALU`. (Only the function variables, but not the type variables, appear explicitly in the parameter list). However, a large generic specification such as the generic specification of complete microprocessor is likely to involve a large number of uninterpreted operators which could lead to an excessively parameterized specification.

Excessive parameterization can be avoided by 'packaging' all of the uninterpreted operators into a single 'representation variable'. Individual operators are 'selected' from the representation variable by previously defined selector functions. A revised version of the generic ALU specification is shown below where the selectors functions `inc`, `add`, `add` and `zero` are used to select individual operators of the representation. Only the representation variable, `rep`, appears explicitly in the parameter list of `ALU`.

```
Define (
  "ALU rep (f0,f1,inp1,inp2,out) =
    ∀t:time.
      out t =
        (((f0 t,f1 t) = (T,T)) ⇒ ((inc rep) (inp2 t)) |
         ((f0 t,f1 t) = (T,F)) ⇒ ((add rep) (inp1 t,inp2 t)) |
         ((f0 t,f1 t) = (F,T)) ⇒ ((sub rep) (inp1 t,inp2 t)) |
                                  (zero rep)")
```

The definitions of the selector functions `inc`, `add`, `sub` and `zero` depend on the 'composition' of the representation variable. If this generic specification of the ALU was a part of a larger specification, then the representation will likely be composed of other operations besides those selected by `add`, `sub`, `inc` and `zero`. For example, the representation variable used in the generic specification of the TAMARACK-3 microprocessor represents thirteen different operations. Phil Windley [38] has described support tools for the HOL system which are helpful when applying this technique of generic specification to large specification problems.

In a hierarchical approach to generic specification, a common representation variable can be passed downwards level by level. For example, the top-level architectural specification of the TAMARACK-3 microprocessor is parameterized by the representation variable `rep` which is passed down to the next lower level, namely, the architectural specification of the control unit and datapath. In turn, these levels of specification pass the representation variable down to even lower levels. This is illustrated below by the definitions of `TamarackImp` and `DataPath` which comprise one branch of the specification hierarchy down to the level of register-transfer level primitives including the ALU.

```
Define (
  "TamarackImp rep
    (datain,dack,idle,ireq,mpc,mar,pc,
     acc,ir,rtn,arg,buf,iack,dataout,wmem,dreq,addr) =
    ∃zeroflag opc cntls.
      CntlUnit rep (dack,idle,ireq,iack,zeroflag,opc,mpc,cntls) ∧
      DataPath rep (
        cntls,datain,mar,pc,acc,ir,rtn,iack,
        arg,buf,dataout,wmem,dreq,addr,zeroflag,opc)")

Define (
  "DataPath rep
    (cntls,datain,mar,pc,acc,ir,rtn,iack,
     arg,buf,dataout,wmem,dreq,addr,zeroflag,opc) =
    ∃bus busokay alu pwr gnd rmem wmar wpc rpc
     wacc racc wir rir wrtn rrtn warg alu0 alu1 rbuf.
      DecodeCntls (
        cntls,
        wmem,rmem,wmar,wpc,rpc,wacc,racc,
        wir,rir,wrtn,rrtn,warg,alu0,alu1,rbuf) ∧
      BusOkay (rmem,rpc,racc,rir,rrtn,rbuf,busokay) ∧
      Interface rep (busokay,wmem,rmem,bus,datain,dataout) ∧
      OR (wmem,rmem,dreq) ∧
      Register (busokay,wmar,gnd,bus,bus,mar) ∧
      AddrField rep (mar,addr) ∧
      Register (busokay,wpc,rpc,bus,bus,pc) ∧
      Register (busokay,wacc,racc,bus,bus,acc) ∧
      TestZero rep (acc,zeroflag) ∧
      Register (busokay,wir,rir,bus,bus,ir) ∧
      OpcField rep (ir,opc) ∧
      Register (busokay,wrtn,rrtn,bus,bus,rtn) ∧
      JKFF (wrtn,rrtn,iack) ∧
      Register (busokay,warg,gnd,bus,bus,arg) ∧
      ALU rep (alu0,alu1,arg,bus,alu) ∧
      Register (busokay,pwr,rbuf,alu,bus,buf) ∧
      PWR pwr ∧
      GND gnd")
```

There are unmistakable indications that large scale verification efforts of the future will depend on support for generic specifications. In a 1989 NASA-sponsored review of the Viper microprocessor verification project, David Musser [32] wrote:

> *"The major weakness of HOL appears to be the lack of effective support for constructing specifications and proofs at a high level of abstraction, ..."*

and more specifically,

> *"HOL is also lacking in packaging features that would help in structuring large specifications, such as those of VIPER or more complex microprocessors"*

While the HOL logic does not provide explicit constructs to support generic specification (as Musser may have hoped to see), it is clear that HOL logic (along with any any other formulation of higher-order logic) has the innate capability to support generic specification.

Although generic specification falls naturally within the domain of higher-order logic, it is admittedly possible to this add this capability to a less expressive language by introducing special constructs for generic specification. An example is the OBJ (first-order equational logic with second-order universal quantification) which provides support for 'parameterized modules' [13].

While special constructs may be included in a specification language to provide a capability for generic specification, this has the disadvantage of requiring additional proof rules needed to manipulate specifications involving these special constructs. We believe, in general, that the introduction of special constructs and proof rules should be avoided if possible. Their introduction is likely to increase the difficulty of implementing a reliable verification system, i.e., a system which ensures proof security. The introduction of special constructs to support generic specification also lessens the possibility of being able to reproduce verification results in other verification systems - whereas the specification techniques outlined in this paper should be applicable in any higher-order specification language.

4 Embedded Formalisms

Our second main reason for claiming that higher-order logic is a good formalism for specifying and verifying hardware is the ability to embed special-purpose formalisms in higher-order logic. Diverse aspects of hardware behaviour are described informally in an equally diverse range of styles: finite-state machines, pseudo-code, logic expressions, timing diagrams are some common examples. However, the conventional approach to hardware specification is, in general, to re-cast these diverse forms of informal description into one basic mold. For example, the specification style advocated by Bevier et al. [3] adheres rigorously to a fixed-format based on self-recursive functions (in Boyer-Moore Logic).

We advocate a different approach, namely, to embed *natural notations* from well-established formalisms such as temporal logic in the framework of higher-order logic. This approach benefits from the built-in economy of these special-purpose notations when they are applied to particular areas of formal description. These benefits include improved specifications (more concise and easier to read) and ease of proof (more powerful proof rules).

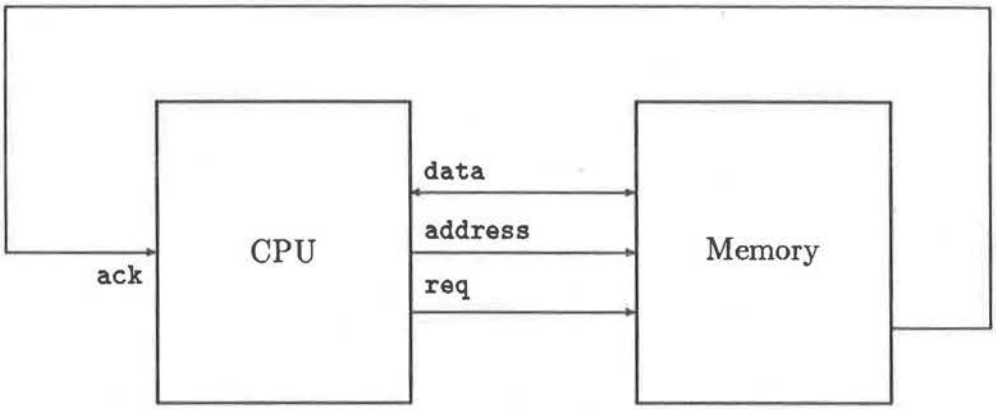

Figure 1: Asynchronous Microprocessor-Memory Interface

The advantages of an embedded formalism are illustrated in this section by the use of temporal logic operators to (partially) specify the asynchronous interaction of a microprocessor with an external memory device using a handshaking protocol. Figure 1 shows the inter-connection of a microprocessor (CPU) with an external memory device. The request and acknowledge signals, `req` and `ack`, are used to synchronize the transfer of data between the microprocessor and external memory. A timing diagram, such as the one shown in Figure 2, is the standard method used by engineers to informally describe how the transfer of data between the two devices is synchronized using `req` and `ack`. The timing diagram describes constraints on the relative order of events which must be satisfied for the transfer of data to be properly synchronized. These constraints may also be expressed in natural language as shown below:

"whenever the request signal becomes true, it must remain true until it is acknowledged"

"every request must eventually be acknowledged"

"whenever the acknowledgement signal becomes true, it must remain true until the request signal returns to false"

"the request signal will eventually return to false after the request is acknowledged"

"whenever the request signal is false, it will remain false until the acknowledgement signal is also false"

"the acknowledgement signal will eventually return to false after the request signal returns to false"

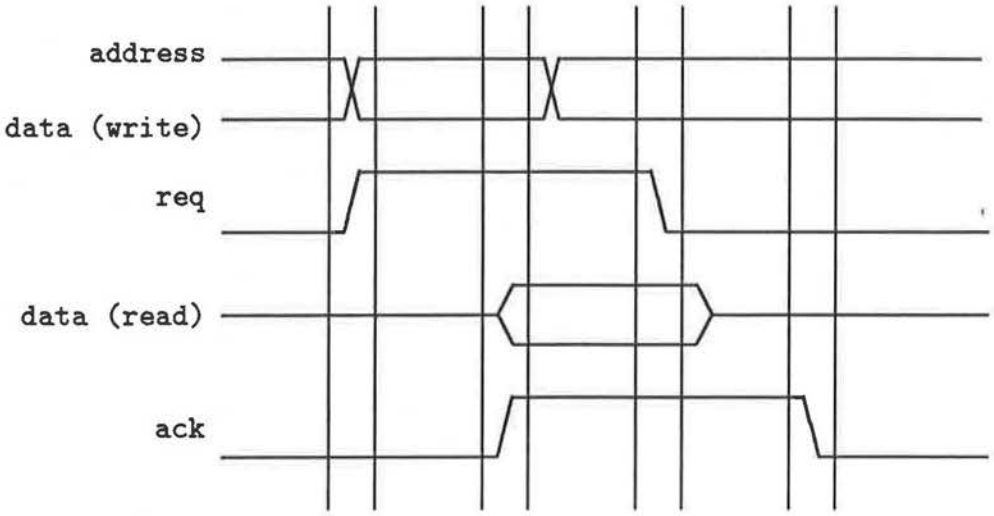

Figure 2: Simplified Timing Diagram for Handshaking Signals

"once false, the acknowledgement signal will remain false until there is a request"

"whenever the acknowledgement signal is false, there will eventually be a request"

We first consider the uninspired approach of translating the above assertions directly into standard predicate calculus notation (these are just 'first-order' formulae).

$$\forall t.\ \mathtt{req}\ t \Longrightarrow (\forall n.\ (\forall m.\ m < n \Longrightarrow \neg \mathtt{ack}(t + m)) \Longrightarrow \mathtt{req}(t + n))$$

$$\forall t.\ \mathtt{req}\ t \Longrightarrow (\exists n.\ \mathtt{ack}(t + n))$$

$$\forall t.\ \mathtt{ack}\ t \Longrightarrow (\forall n.\ (\forall m.\ m < n \Longrightarrow \mathtt{req}(t + m)) \Longrightarrow \mathtt{ack}(t + n))$$

$$\forall t.\ \mathtt{ack}\ t \Longrightarrow (\exists n.\ \neg \mathtt{req}(t + n))$$

$$\forall t.\ \neg \mathtt{req}\ t \Longrightarrow (\forall n.\ (\forall m.\ m < n \Longrightarrow \mathtt{ack}(t + m)) \Longrightarrow \neg \mathtt{req}(t + n))$$

$$\forall t.\ \neg \mathtt{req}\ t \Longrightarrow (\exists n.\ \neg \mathtt{ack}(t + n))$$

$$\forall t.\ \neg \mathtt{ack}\ t \Longrightarrow (\forall n.\ (\forall m.\ m < n \Longrightarrow \neg \mathtt{req}(t + m)) \Longrightarrow \neg \mathtt{ack}(t + n))$$

$$\forall t.\ \neg \mathtt{ack}\ t \Longrightarrow (\exists n.\ \mathtt{req}(t + n))$$

The above set of logical formulae accurately capture the handshaking protocol (as informally described by both the timing diagram and the set of natural language assertions). However, these formulae are neither concise nor easy to read (and understand). For example, the formula,

$$\forall t.\ \mathtt{req}\ t \Longrightarrow (\forall n.\ (\forall m.\ m < n \Longrightarrow \neg \mathtt{ack}(t + m)) \Longrightarrow \mathtt{req}(t + n))$$

is supposed to say:

"whenever the request signal becomes true, it must remain true until it is acknowledged"

However, most people would find it difficult to translate between this natural language description and the logical formulae shown above.

In contrast, the elegant notation of temporal logic can be used to write a concise and easy to read specification of the handshaking protocol. For example, the constraint,

"whenever the request signal becomes true, it must remain true until it is acknowledged"

translates easily into the formulae,

$$(\mathtt{req} \longrightarrow (\mathtt{req}\ \mathcal{U}\ \mathtt{ack}))$$

where $\longrightarrow$ and $\mathcal{U}$ may be read as "implies" and "until".

Similarly, instead of,

$$\forall t.\ \mathtt{req}\ t \Longrightarrow (\exists n.\ \mathtt{ack}(t + n))$$

we can use the notation of temporal logic to express this same condition,

$$(\mathtt{req} \longrightarrow (\Diamond \mathtt{ack}))$$

"every request must eventually be acknowledged"

where '$\Diamond$' can be read as "eventually".

The notation of temporal logic can be used in a similar manner to formalize the other six natural language assertions describing the handshaking protocol.

$$(\texttt{ack} \longrightarrow (\texttt{ack} \cup (\sim\texttt{req})))$$

$$(\texttt{ack} \longrightarrow (\Diamond(\sim\texttt{req})))$$

$$((\sim\texttt{req}) \longrightarrow ((\sim\texttt{req}) \cup (\sim\texttt{ack})))$$

$$((\sim\texttt{req}) \longrightarrow (\Diamond(\sim\texttt{ack})))$$

$$((\sim\texttt{ack}) \longrightarrow ((\sim\texttt{ack}) \cup \texttt{req}))$$

$$((\sim\texttt{ack}) \longrightarrow (\Diamond\texttt{req}))$$

The use of temporal logic operators to specify handshaking protocols is well known from previous work [4, 11, 19]. Indeed, temporal logic has been used in a more general way as a hardware specification language [30, 31]. However, our interest lies in the use of temporal logic notation alongside other specialized notations within the context of large, many-sided specification problems. For example, the verified computing system described in [27], which includes both software and hardware levels, uses temporal logic to specify the asynchronous interface of a verified microprocessor as well as notation from denotational semantics to specify the semantics of a simple programming language.

When a variety of specialized notations are used in a large specification effort, we can ensure that the sum of the specification parts results in a unified whole by embedding the specialized notations into the common framework of higher-order logic. Strictly speaking, each notation can be regarded as high-level notation for higher-order logic. In the HOL formulation of higher-order logic, this can be done *safely* in a definitional manner where:

1. Primitive operators are defined as higher-order functions.
2. Transformation rules are derived as theorems of higher-order logic (as consequences of the operator definitions).

For example, a useful set of temporal logic operators (including those already mentioned) can be defined as follows:

```
Define ("□P = λt. ∀n. P (t+n)")

Define ("◇P = λt. ∃n. P (t+n)")

Define ("◯P = λt. P (t+1)")

Define ("P ∪ Q = λt. ∀n. (∀m. m < n ⟹ ¬(Q (t+m))) ⟹ P (t+n)")

Define ("∼P = λt. ¬(P t)")

Define ("P and Q = λt. P t ∧ Q t")

Define ("P ⟶ Q = λt. P t ⟹ Q t")
```

Assertions in temporal logic are predicates of time. To convert temporal logic assertions into assertion of higher-order logic, we introduce a simple notion of validity[2] where a temporal logic assertion is valid if and only if it is true at all times.

```
Define ("VALID P = ∀t. P t")
```

Improved specifications are not the only potential benefit of using a more specialized notation such as temporal logic; another potential benefit is the introduction of powerful transformation rules contributing to a considerable reduction in overall proof effort. For instance, the transformation rule,

$$\frac{\texttt{P and } \sim\texttt{Q} \longrightarrow \bigcirc\texttt{P}}{\texttt{P} \longrightarrow (\texttt{P} \cup \texttt{Q})}$$

can be used to achieve in a single proof step what would otherwise require a relatively complex sequence of low-level proof steps. Transformations can be *safely* introduced as theorems of higher-order logic (as logical consequences of the operator definitions). For instance, the above transformation rule is expressed by the following theorem of higher-order logic:

$$\forall \texttt{P Q. VALID } ((\texttt{P and } \sim\texttt{Q}) \longrightarrow \bigcirc\texttt{P}) \Longrightarrow \texttt{VALID}(\texttt{P} \longrightarrow (\texttt{P} \cup \texttt{Q}))$$

[2] Although this definition of validity allows us to express temporal logic assertions in higher-order logic (and is essentially the same as the approach used by Hale [20]), it does not capture the 'whole meaning' of validity in a model-theoretic sense [21].

The introduction of transformation rules in this manner is *safe* because they are merely consequences of the operator definitions. In the case of HOL logic, constraints on the introduction of definitions (in general) prevent the possibility of inconsistency arising from the definitions of these operators [17].

Higher-order logic is a natural choice as a general framework for supporting embedded notations. This is easily seen in the case of temporal logic. Temporal logic formulae are naturally represented as predicates of time. Temporal logic operators are naturally represented as higher-order functions by regarding them as functions which are applied to formulae (i.e., predicates of time) to yield new formulae.

The definitional approach to embedding a specialized notation in a verification system such as the HOL system contrasts with syntax-based approaches. For example, the Cornell Synthesizer Generator [34] can be used to build support tools for specialized notations. In a syntax-based approach, transformation rules are simply 'programmed' (perhaps inconsistently) into the system. This contrasts with a definitional approach where transformation rules are derived as logical consequences of a semantic theory (i.e., the set of primitive operator definitions).

The embedding of specialized notations in higher-order logic is currently a very active area of research within the HOL community. This includes work on: programming logics [16]; temporal logics [20, 25]; and process algebras such as Hoare's CSP [5] and Milner's π-calculus [18].

5 Arguments Against Using Higher-Order Logic

The use of higher-order logic for specifying and verifying hardware is sometimes criticized as impractical or extravagant. These criticisms often have the flavor of:

> *"The logic of higher-order functions is difficult, and in particular, higher order unification is undecidable. Moreover, higher order expressions are notoriously difficult for humans to read and write correctly."*

These concerns about undecidability and difficult notation seem, in practice, to be either unimportant or unaccountable. Even for approaches that place considerable emphasis on proof automation, experience shows that higher-order unification is required only infrequently to perform higher-order proofs [33]. While certain notations for higher-order logic may be less readable than others, it seems unaccountable to suggest that all such notations are inherently difficult to read. Indeed, a major point in this paper is to argue that the expressibility of higher-order logic can be used to improve the readability of a formal specification by means of an embedded notation.

Sometimes it is argued that formal proofs in higher-order logic are *generally* more difficult than formal proofs in other formalisms such as first-order logic. It is true that some specialized forms of proof, for instance, those involving Hilbert's ε-operator, can be difficult. But these specialized forms of proof are not frequently encountered, if at all, in most practical verification work - and such proofs generally yield highly re-usable theorems. Many practically-oriented hardware proofs in higher-order logic are similar in difficulty to proofs in first-order logic.

Another criticism is that the use of higher-order logic as a hardware specification language is extravagant. This is sometimes argued by showing how simple examples of hardware specification can be rendered in less expressive formalism, e.g., the formal specification of an n-bit ripple-carry adder. However, simple examples of hardware verification cannot test the adequacy of a hardware specification language in any reasonable way. A better test of the adequacy of a hardware description language is a problem such as the one encountered by Warren Hunt [24] when specifying the FM8501 microprocessor without the benefit of existential quantification. The absence of existential quantification in Boyer-Moore Logic made it difficult to specify the unknown duration of delays in asynchronous handshaking interactions of the FM8501 with external memory. Great interest has been shown in this particular problem of specifying asynchronous interfaces by number of researchers [10, 27, 35, 36].

6 Summary

This paper has characterized conventional approaches to hardware specification in two main ways:

Closed-World: Every symbol which appears in a specification is fully defined even when the definitions of some of these symbols are not relevant to the verification problem.

Fixed Format: Diverse forms of informal behavioural description (e.g., timing diagrams, flowcharts, pseudo-code, block diagrams) are re-cast into one fixed format pre-determined by the specification language.

This conventional approach has been successfully applied in many examples of hardware verifications, notably, the impressive work by researchers at Computational Logic, Inc., on the formal verification of a complete computing system. The formal specification of the Viper microprocessor was also rendered as essentially a closed-world, fixed-format specification.

In spite of the success thus far of these conventional approaches, we advocate an alternative approach characterized by:

Generic Specification: The explicit or implicit parameterization of formal specifications to the greatest possible extent without changing the essence of the verification problem.

Embedded Formalisms: The embedding of diverse notations in a common framework to support formal specifications of an equally diverse range of behaviours.

We believe that approaches based on generic specification and the use of embedded notations will be more likely to succeed as verification problems grow in complexity beyond the current state-of-the-art examples. The same forces that lead to the emergence of generic software description will also contribute to the emergence of generic hardware description. The advantages of improved specification and easier proofs resulting from the use of natural notations such as temporal logic will prevail over the limitations of fixed-format specifications. Because higher-order logic naturally supports both generic specification and the use of natural notations, we believe that higher-order approaches will contribute significantly to future developments in formal hardware verification.

Acknowledgements

Several people have directly contributed to the ideas presented in this paper. Mike Gordon (of Cambridge University) first urged me to consider the use of temporal logic operators to improve upon earlier attempts to specify handshaking protocols in higher-order logic. Discussions with John Rushby and his colleagues at SRI International prompted me to think about the advantages of specifying hardware generically. I have also benefitted from recent discussions about temporal logic with Amit Jasuja (of UC Davis) and about generic specification with John Van Tassel (of Cambridge University) and Phil Windley (of the University of Idaho). This research is currently funded by an NSERC Operating Grant from the Canadian Government.

References

[1] F. Anceau, The Architecture of Microprocessors, Addison-Wesley Publishing Company, Wokingham, 1986.

[2] David A. Basin and Geoffrey M. Brown and Miriam E. Leeser, Formally Verified Synthesis of Combinational CMOS Circuits, in: L. Claesen, ed., Formal VLSI Specification and Synthesis, North-Holland, 1990, pp. 197-206.

[3] W. Bevier, W. Hunt, J Moore, and W. Young, An Approach to Systems Verification, Journal of Automated Reasoning, Vol. 5, No. 4, November 1989. Also Report No. 41, Computational Logic, Inc., Austin, Texas, April 1989.

[4] G. Bochman, Hardware Specification with Temporal Logic, IEEE Transactions on Computers, Vol. C-31, No. 3, March 1982, pp. 223-231.

[5] Albert John Camilleri, Mechanizing CSP Trace Theory in Higher Order Logic, IEEE Transactions on Software Engineering, Vol. SE-16, No. 9, September 1990, pp. 993-1104.

[6] Rachel Cardell-Oliver, The Specification and Verification of Sliding Window Protocols in Higher Order Logic, Report No. 183, Computer Laboratory, Cambridge University, October 1989.

[7] Avra Cohn, A Proof of Correctness of the Viper Microprocessor: The First Level, in: G. Birtwistle and P. Subrahmanyam, eds., VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, Boston, 1988, pp. 27-71. Also Report No. 104, Computer Laboratory, Cambridge University, January 1987.

[8] Avra Cohn, Correctness Properties of the Viper Block Model: The Second Level, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989, pp. 1-91. Also Report No. 134, Computer Laboratory, Cambridge University, May 1988.

[9] Avra Cohn, The Notion of Proof in Hardware Verification, Journal of Automated Reasoning, Vol. 5, May 1989, pp. 127-139.

[10] Stephen D. Crocker, Eve Cohen, Sue Landauer and Hilarie Orman, Reverification of a Microprocessor, Proceedings of the 1988 IEEE Symposium on Security and Privacy, 18-21 April 1988, Oakland, California Computer Society Press, Washington, D.C., 1988, pp. 166-176.

[11] D. Dill and E. Clarke, Automatic Verification of Asynchronous Circuits using Temporal Logic, IEE Proceedings, Vol. 133, Pt. E, No. 5, September 1986, pp. 276-282.

[12] H. Eveking, How to Design Correct Hardware and Know It. G. Milne, ed., The Fusion of Hardware Design and Verification, Proceedings of the IFIP WG 10.2 International Working Conference, Glasgow, Scotland, 3-6 July 1988, North-Holland, 1988, pp. 250-262.

[13] Joseph A. Goguen, OBJ as a Theorem Prover with Applications to Hardware Verification, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989, pp. 219-267. Also Report No. SRI-CSL-4R2, Computer Science Laboratory, SRI International, Menlo Park, August 1988.

[14] Joseph A. Goguen, Higher Order Functions Considered Unnecessary for Higher Order Programming, Report No. SRI-CSLL-88-1R, Computer Science Laboratory, SRI International, Menlo Park, January 1988.

[15] M. J. C. Gordon, Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware, in: G. Milne and P. Subrahmanyam, eds., Formal Aspects of VLSI Design, Proceedings of the 1985 Edinburgh Conference on VLSI, North-Holland, 1986, pp. 153-177.

[16] Michael J. C. Gordon, Mechanizing Programming Logics in Higher Order Logic, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989, pp. 387-439. Also Report No. 145, Computer Laboratory, Cambridge University, September 1988.

[17] Michael J. C. Gordon et al., The HOL System Description, Cambridge Research Centre, SRI International, Suite 23, Miller's Yard, Cambridge CB2 1RQ, England.

[18] Michael J. C. Gordon and Thomas F. Melham, Presentation at the 1990 HOL User's Group Meeting, Aarhus, 1-2 October 1990.

[19] M. Fujita, H. Tanaka and T. Moto-oka., Temporal Logic Based Hardware Description and Its Verification with Prolog, New Generation Computing, No. 1, 1983, pp. 195-203.

[20] Roger W. S. Hale, Programming in Temporal Logic, Ph.D. Thesis, Report No. 173, Computer Laboratory, Cambridge University, July 1989.

[21] Roger Hale, private communication, 1989.

[22] F. K. Hanna and N. Daeche, Specification and Verification of Digital Systems using Higher-Order Predicate Logic, IEE Proceedings, Vol. 133, Part E, No. 5, September 1986, pp. 242-254.

[23] F. K. Hanna, N. Daeche and M. Longley, VERITAS$^+$: A Specification Language Based on Type Theory, in: M. Leeser and G. Brown, eds., Specification, Verification and Synthesis: Mathematical Aspects, Proceedings of a Workshop, 5-7 July 1989, Ithaca, N.Y., Springer-Verlag, 1989.

[24] Warren A. Hunt, FM8501, A Verified Microprocessor, Ph.D. Thesis, Report No. 47, Institute for Computing Science, University of Texas, Austin, December 1985.

[25] Amit Jasuja, private communication, 1990.

[26] Jeffrey J. Joyce, Totally Verified Systems: Linking Verified Software to Verified Hardware, in: M. Leeser and G. Brown, eds., Specification, Verification and Synthesis: Mathematical Aspects, Proceedings of a Workshop, 5-7 July 1989, Ithaca, N.Y., Springer-Verlag, 1989. Also Report No. 178, Computer Laboratory, Cambridge University, September 1989.

[27] Jeffrey J. Joyce, Multi-Level Verification of Microprocessor-Based Systems, Ph.D. Thesis, Computer Laboratory, Cambridge University, December 1989. Report No. 195, Computer Laboratory, Cambridge University, May 1990.

[28] Jeffrey J. Joyce, Generic Specification of Digital Hardware, in: M. Sheeran and G. Jones, eds., Proceedings of Workshop on Digital Circuit Correctness, September 1990, Oxford.

[29] Jeffrey J. Joyce and John P. Van Tassel, Fully Generic Description of Hardware in VHDL, (submitted to CHDL 91).

[30] Miriam E. Leeser. Reasoning about the Function and Timing of Integrated Circuits with Prolog and Temporal Logic, Ph.D. Thesis, Computer Laboratory, Report No. 132, Computer Laboratory, Cambridge University, April 1988.

[31] Benjamin Moszkowski, A Temporal Logic for Multilevel Reasoning about Hardware, IEEE Computer Vol. 18, No. 2, February 1985, pp. 10-19.

[32] David R. Musser, Report on the HOL (Higher Order Logic) Proof Checker, A report sponsored in part through a subcontract from Computation Logic, Inc., sponsored in turn by NASA Langley Research Center, AIRLAB Division, through the Defense Advanced Research Agency, ARPA order 584144, November 2, 1989.

[33] John Rushby, private communication, 1990.

[34] T. Reps and T. Tietelbaum, The Synthesizer Generator, Springer-Verlag, 1989.

[35] R. C. Sekar and M. K. Srivas, Formal Verification of a Microprocessor Using Equational Techniques, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989, pp. 171-218.

[36] P. A. Subrahmanyam, What's in a Timing Discipline ?: Considerations in the Specification and Synthesis of Systems with Interacting Asynchronous and Synchronous Components, in: M. Leeser and G. Brown, eds., Specification, Verification and Synthesis: Mathematical Aspects, Proceedings of a Workshop, 5-7 July 1989, Ithaca, N.Y., Springer-Verlag, 1989.

[37] F. W. von Henke, J. S. Crow, R. Lee, J. M. Rushby and R. A. Whitehurst, The EHDM Verification Environment: An Overview, Proceedings of the 11th National Computer Security Conference, Baltimore, October 1988, pp. 147-155.

[38] Phillip J. Windley, The Formal Verification of Generic Interpreters, Ph.D. Thesis, Division of Computer Science, University of California, Davis, Report No. CSE-90-22, July 1990.