Formal Specification and Verification of Hardware: A Comparative Case Study

V. Stavridou, H. Barringer and D.A. Edwards

Department of Computer Science, University of Manchester, UK

Abstract

Formal methods are applied with increased frequency in the specification and verification of digital systems as an alternative to traditional methods of establishing correctness, such as simulation and testing. This article briefly outlines the goals and the philosophy of the *HA*rdware *VE*rification project at the Department of Computer Science within the University of Manchester. Our objective here is to report on the results of a first controlled experiment comparing formalisms and systems that are currently used for formally specifying and verifying both hardware and software systems. Our strategy consists of working with incrementally "harder" test cases, which are used to investigate the characteristics and thus the pros and cons of each formalism. The example used is a purely combinational device.

1 Introduction

The quest for quality, "error-free", software has led to the introduction of mathematical techniques and related tools in the software development process. But error-free software is only as good as the hardware it is running on. This is particularly important in safety critical applications. Furthermore, fully testing and documenting a machine before it comes into the market is a resource-intensive procedure which is often ill-served by traditional methods such as simulation. Hence formal methods have migrated into hardware development, aiming to provide safer and more extensive design and testing procedures. [Bar84, CPC87] argue that verification of circuits by proof is a better and cheaper alternative to testing which in any case is incomplete. [Hun85] contains a substantial verification example (the FM8501 microprocessor) while [Coh87] describes some aspects of the verification of the Viper microprocessor. Although these experiments investigate the functionality of the microprocessors at an abstract level, they do nonetheless illustrate that formal methods can be successfully used on realistic examples.

Our objective in this paper is not to provide justification for the use of formal methods in hardware development. Rather we try to identify a way in which to obtain maximal benefit from the use of such techniques in the specification and verification of hardware systems. We thus attempt to evaluate a set of systems through a case study with two main issues under consideration.

1. Which formalism is most appropriate.
2. How to make best use of the formalism or tool.

2 The HAVE Project

The software and hardware design processes have traditionally been viewed as separate disciplines. However, as the distinctions between hardware and software become further blurred, it becomes apparent that technology transfer between VLSI and software engineering can be beneficial to both fields [SD84]. The objective of the HAVE project is to formulate a design methodology for digital hardware systems based on formal methods. We are interested in such methods whether they have been traditionally used in connection with hardware or software development.

2.1 Using a formal specification

Should a formalism be executable? If yes, what does it mean for a formalism and consequently models in that formalism to be executable? What kind of support should be provided to make best use of the available executability? While it is recognised that the production of a formal specification aids problem understanding and leads to comprehensive design documentation, we feel for the reasons itemized below, that the benefit of using formal methods and tools can be maximised if a notation is enhanced with some form of processability and machine support.

1. If machine support is not available, correctness of an implementation against a specification must be established manually. This is usually impracticable due to the volume of the manipulations required even for trivial examples.
2. Formal proof of correctness is a resource intensive activity. Before formal proof is attempted it is sensible to obtain some degree of faith in the design through simulation. If a model is not executable, direct simulation is not possible.

CH2540-3/88/0000/0197$01.00 © 1988 IEEE

3. There is a great level of investment in existing VLSI/CAD tools in terms of financial and human resources. Formal methods and tools should be integrated or interfaced with such systems in order to provide flexible and productive design environments. If a notation does not possess the element of processability, integration is not possible.

Therefore, we only consider formalisms that provide either directly executable models, or models which can be executed after applying some transformation to them.

We believe that the formal specification of a system serves a *dual* purpose. It is first used for simulation purposes in order to ensure that it exhibits the desired behaviour. Once a fair degree of certainty about the suitability of the design is attained, a suitable theorem prover is used to make formal checks on the specification. In general, such checks involve proving that a certain implementation *implies* either the specification or some desired property of the specification. The specification, implementation and properties are expressed as *assertions* in the given logic. Specific instances of proofs include facts such as:

- A high level (abstract) design is *equivalent* to the structural composition of lower level (concrete) components, for example that an incrementer, a multiplexer and some register primitives when wired up in a specific way, produce the behaviour of a loadable counter.
- An implementation *implies* the specification, which is a generalisation of equivalence proofs. For example, *if* an incrementer, a multiplexer and some register primitives are wired up in a specific way, *then* the resulting circuit will behave as a loadable counter.
- A specification *implies* a property that the system should exhibit, for instance that a binary adder does indeed produce the mathematical sum of its inputs.

A consequence of the above approach is that during the evaluation stage, formalisms are considered both on an individual basis and also as pairs (*specification language, theorem prover*). This second consideration is important from the point of view of providing an integrated design support environment.

2.2 Which formalism?

Most systems aim to provide a model or tool tailored to a specific level of abstraction. There is no panacea for all levels of abstraction involved in the design and realisation of a piece of hardware. It is, therefore, the case that the best formalism to use depends on the nature of the problem under investigation. At this stage, the HAVE project is concerned with synchronous systems only. The problems are examined from the register transfer level down to gate level. This option circumvents the difficulties of working with transistor level designs, since currently available formal models are highly simplified in comparison to the physical operation of the hardware devices themselves. Work aiming towards the production of adequate formal models of hardware at the transistor level, is described in [Win86] and [MNP87].

There is a plethora of formalisms which have been proposed in association with formal specification and verification of hardware systems. It therefore useful to impose some structure on this set in order to facilitate comparisons of like with like. Although this is a fuzzy distinction, for classification purposes, we can view formalisms as *modelling* (where we construct a model of the system under investigation) or *behavioural* (where we define a set of properties that the system's behaviour should possess). The formalisms under investigation include various logics such as PPLAMBDA [Pau83] which is supported by the LCF proof checker [GMW79], the Boyer-Moore logic which is mechanised by the synonymous theorem prover [BM83], higher order logic as implemented by the HOL [Gor85] and Veritas [HD85] systems, equational logic which is supported by the REVE system [Les83] and finally temporal logics as described in [Mos83, CES83]. We also consider specific calculi such as the Calculus of Communicating Systems [Mil80] and Circal [Mil83]. Finally, we examine special purpose hardware description languages such as LTS [BFM85] and ELLA [ELL86], as well as general purpose programming languages and variants, for example Lisp, Standard ML [Mil84], Hope [BMS80], OBJ [FGJM85], μFP [She84] and νFP [PSE85]. The formalisms named here are representative but the list is by no means exhaustive.

Once the level of abstraction at which the system will be treated has been identified, the most suitable formalism available must be chosen. It is very difficult to make a rational choice based solely on information available from the literature. Practical experimentation with various systems, on the other hand, provides useful insight and experience, which affords the grounds for a rational choice to be made. The HAVE approach is based on a detailed, comparative study of alternative approaches, using examples of increasing complexity. A careful distinction must be drawn between such test cases and the establishment of benchmarks. The various notations and systems have historically evolved with a particular level of abstraction in mind. It is not, therefore, straightforward or indeed possible to establish universal criteria of *goodness* [CP87]. The test cases used here are not meant to be taken as benchmarks. Here is the list of the chosen examples:

1. n-bit wide parallel binary adder (purely combinational logic).
2. Twisted ring counter (state transition machine).
3. PAL control logic for PCB/VLSI routing engine [SE87] (substantial state transition machine).
4. Interfaces between modules of the Manchester Dataflow prototype [GKB87] (composition of state transition machines, also involving multiple clocks).

The first two examples were chosen for simplicity, while the remaining two were chosen in order to investigate the applicability of notations in real engineering problems. In this paper, we report on the results of the first test case. The interesting aspects of the problem determine the set of notations that can be sensibly investigated. For instance, temporal logic is

clearly not an appropriate formalism to try for the n-bit adder, as the temporal aspects of the circuit's behaviour are not of interest. The list of systems that have been investigated using the parallel adder are:

1. Specification Languages

   OBJ [FGJM85], Lisp, Standard ML [Mil84], Hope [BMS80], μFP [She84], νFP [PSE85], ELLA [ELL86], LTS [BFM85].

2. Theorem Provers

   REVE [Les83], Boyer-Moore [BM83], Cambridge LCF [Pau85], HOL [Gor85].

The classification of Lisp, Standard ML and Hope above as specification languages, reflects their role in our approach. For reasons similar to those relating to the use of declarative instead of imperative languages for software development [Bac78], we mainly consider the former.

The next section presents the n-bit adder example in sufficient detail to enable specification and verification of its properties.

3 The Example

This case study is based on a simple, combinational circuit, namely the n-bit wide parallel binary adder. Apart from simulating the circuit, the aim is to prove that the circuit produces the mathematical sum of its inputs. This is a property proof. The following factors were taken into account when choosing the adder.

- The circuit is combinational, hence timing issues do not have to be considered, thus simplifying the exercise.
- The circuit displays a hierarchical structure thus exploring the composition/decomposition support provided by the various formalisms.
- The proof of the desired property requires inductive arguments.

Below, we show the definitions of the various components of the adder together with some functions necessary for the proof. These descriptions are shown in Lisp.

The basic building block is the half adder which consists of an AND and an XOR gate:

```
(defun halfadder(x y)
  (list (xor x y) (and x y)))
```

The full adder is made up of two half adders and an OR gate:

```
(defun fulladder(x y c-in)
  (list (sum (halfadder (sum (halfadder x y)) c-in ))
        (or (carry (halfadder x y))
            (carry (halfadder (sum (halfadder x y)))))))
```

where

```
(defun sum(pair) (car pair))
(defun carry(pair) (cdr pair))
```

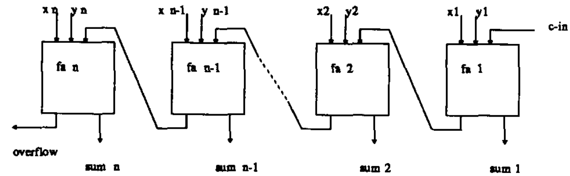

Figure 1: N-bit wide parallel binary adder

The n-bit adder consists of a cascade of n full adders with the output carry of the $(i-1)^{th}$ unit connected to the input carry of the ith unit as shown in Figure 1.

The functions below implement mappings from lists of bits onto natural numbers and vice versa. To eliminate the need for traversing the list before additions, the head of the list contains the least significant bit:

```
(defun bin-to-nat(x)
  (if (equal x nil)
      0
      (plus (bit (car x))
            (times 2 (bin-to-nat (cdr x))))))
```

where function `bit` provides a mapping from bits to natural numbers:

```
(defun bit(x) (if x 1 0))
```

`nat-to-bin` maps natural numbers onto lists of bits:

```
(defun nat-to-bin(x)
  (if (equal x 0)
      nil
      (cons (not (evenp x))
            (nat-to-bin (quotient x 2)))))
```

Finally, the function `inc` adjusts the input list of bits so that its natural number representation is incremented by 1:

```
(defun inc(x)
  (if (equal x nil)
      (cons t nil)
      (if (car x)
          (cons nil (inc (cdr x)))
          (cons t (cdr x)))))
```

4 The Case Study

As outlined above, it is the philosophy of the HAVE group to use an executable specification in two ways:

- simulation by executing it with suitable inputs, and
- formal verification using theorem provers.

In this section, we present the results of the case study, involving four pairs of ⟨*specification language, theorem prover*⟩. The same simulation tests are used for all specification languages, and the theorem provers attempt to prove the same theorem. In general, the specification text requires amendments before

it can be input to the theorem prover, because the components of our pairs were not designed as integrated environments. The theorem provers under investigation support different logics and associated inference mechanisms. They also represent radically different approaches to theorem proving, ranging from highly interactive proof construction to automatic proofs. An attempt is made below, to present these systems in a unified framework.

4.1 Lisp and Boyer-Moore

Lisp was the first so called functional language to gain wide acceptance as a general purpose programming language. Characteristics of interest for this study are weak typing, higher order functions and efficient implementations. The specification of the adder is as follows. The definition is factored into two functions in order to enhance readability.

```
(defun adder(c-in x-bits y-bits)
  (if (or (equal x-bits nil)
          (equal y-bits nil))
      basic-adder(x-bits y-bits c-in)
      (cons (sum (fulladder(car x-bits)
                           (car y-bits)
                           c-in))
            (adder (carry (fulladder (car x-bits)
                                     (car y-bits)
                                     c-in))
                   (cdr x-bits)
                   (cdr y-bits)))))


(defun basic-adder(x-bits y-bits c-in)
  (if c-in
      (if (equal x-bits nil)
          (inc y-bits)
          (inc x-bits))
      (if (equal x-bits nil)
          y-bits
          x-bits)))
```

Simulation is effected using the function `test` as defined below. Here, in common with the rest of the languages, high and low voltages on lines are modelled using the boolean values `t` and `nil` respectively. The input carry is set to `nil` to indicate that an addition is about to take place.

```
(defun test(n1 n2)
  (patom n1) (patom " + ") (patom n2)
  (patom " = ")
  (patom (bin-to-nat (adder nil
                            (nat-to-bin n1)
                            (nat-to-bin n2))))
  (terpri))

(test 2213 21)
;2213 + 21 = 2234
```

Similarly with [Hun85], we use Lisp in conjunction with the Boyer-Moore theorem prover. This is an automatic theorem prover in the sense that once the proof of a theorem has been invoked, the user can no longer interfere. The Boyer-Moore logic is a quantifier-free first order logic with equality and the input language is a variant of pure Lisp. The inference rules consist of propositional calculus and associated inference rules, the principle of Noetherian induction and instantiation. The proof process consists of creating an environment containing the relevant definitions (which must satisfy a *well-founded* ordering) and invoking proof of various necessary lemmas until the target theorem can be proved. Discovering which lemmas are necessary is not trivial for most proofs. The property which the behaviour of the adder should possess is expressed by the following proposition:

```
(prove-lemma correct-adder (rewrite)
  (implies (and (bitvp x-bits)
                (bitvp y-bits)
                (bitvp c-in))
           (equal
             (bin-to-nat (adder c-in x-bits y-bits))
             (plus (bin-to-nat x-bits)
                   (bin-to-nat y-bits)
                   (bit c-in)))))
```

where `bitvp` is a unary predicate which is true when its argument is a valid bit vector and `rewrite` shows that after the theorem has been proven it can be used in subsequent proofs as a rewrite rule.

The proof of this theorem essentially consists of a two-layered inductive argument on the variables `x-bits` and `y-bits`. The powerful built-in induction heuristics of the theorem prover make the proof trivial. However, since induction is just one inference rule this result cannot be generalised. In fact, specifying and reasoning about behaviours through time require more expressive power than the propositional calculus affords, in the sense that such behaviours are modelled naturally by higher order functions. In [Hun85] time is modelled by introducing an extra list argument to recursive functions describing behaviour in time. Every time the function calls itself, a `car` operation is applied to the list argument, thus emulating a clock tick. This time model is not entirely satisfactory. In general:

1. Inductive proofs are straightforward once the necessary set of lemmas has been identified.

2. Intuition and experience are needed in order to *train* the theorem prover for particular tasks, in the face of no user interaction.

3. The output of the system (46 pages in this case) is very verbose thus further accentuating the above problems.

Boyer-Moore was by far the most efficient system for this particular example. Discovering the required lemmas, proving them and finally proving the adder theorem itself took half a man day.

4.2 OBJ and REVE

OBJ [FGJM85] is a high level specification language based on the concept of *abstract data types* and equational logic. Its operational semantics are given by (conditional) rewrite rules. OBJ has traditionally been used for writing and executing algebraic specifications of programs. Characteristic features of the language include:

- Comprehensive parameterisation support which aids modularity and provides implicit higher order function capabilities.

- Mixfix syntax which enhances readability.

- Term rewriting operational semantics which, in the absence of timing constraints, can be used for circuit optimisation.

- Function definition through case analysis eliminating the need for selector functions (e.g. car and cdr in Lisp) which give rise to strange encodings.

The specification of the adder is shown below:

```
OBJ Adder / Vector Fulladder
OPS adder : BOOL vector vector -> vector
VARS bit1, bit2, c-in : BOOL
     v, rest1, rest2 : vector
EQNS
   ( adder(T,v,nil) = inc(v) )
   ( adder(F,v,nil) = v )
   ( adder(T,nil,v) = inc(v) )
   ( adder(F,nil,v) = v )
   ( adder(c-in, bit1.rest1, bit2.rest2) =
         sum(fulladder(bit1,bit2,c-in)).
         adder(carry(fulladder(bit1,bit2,c-in)),
               rest1,rest2) )
JBO
```

where the imported objects Vector and Fulladder, respectively introduce binary vectors and the specification of the full adder. The usual *cons* operation is shown as `_._`. Voltages on lines are modelled using the booleans `T` and `F`. The simulation test cases were of the form `test(n1,n2)` where `test` is defined as

```
test(n1,n2) =
bin-to-nat(adder(F,nat-to-bin(n1),nat-to-bin(n2)))
```

where the functions `bin-to-nat` and `nat-to-bin` are as defined in section 3. A typical simulation test is:

```
> run test(2213,21) nur
AS nat : 2234
```

The theorem prover used in conjunction with OBJ is REVE [Les83]. REVE contains an implementation of the Knuth-Bendix completion procedure modulo associative-commutative operators and procedure supporting Inductionless Induction [HH82] or proof by consistency. Thus, proving a theorem involves compiling a set of equations (the rewrite rules of the OBJ specification) into a confluent set of rewrite rules and then using standard equational reasoning (that is substitution of equals by equals) for theorems involving ground (no variables) terms only, or invoking the inductionless induction process for theorems which contain variables. This is the proposition stating that the adder does indeed produce the mathematical sum of its inputs:

```
bin-to-nat(adder(c-in,x,y)) ==
bin-to-nat(x) + bin-to-nat(y) + bit(c-in)
```

REVE is an automatic theorem prover in the sense that once the proving process (completion procedure) has been initiated, the user has very little influence over the course of the proof. In the case of REVE, the only user interaction is the ability to choose between alternative operator orderings. An implication of this is that when a proof fails there is little or no indication about the reason.

Although REVE produced the shortest proof trace (4 pages) compared with the rest of the systems, it is considered unsatisfactory for the following reasons:

1. The version used (2.4) does not include a completion procedure for conditional equations, thus imposing severe constraints on the class of problems that can be tackled.

2. More fundamentally, compiling a set of equations into a confluent set of rewrite rules is a partially solvable problem, that is the algorithm may fail to terminate with failure for non theorems. This fact, coupled with inefficient implementations of the completion procedure [Sta87a] , further limits the scope of the approach.

3. Finally, REVE is single-sorted, whereas the OBJ input is many-sorted. This requires further adjustments of the OBJ input and, in addition, allows rewriting of ill-typed terms.

It took 3 man weeks to prove the adder theorem with REVE. Most of this time was spent in trying to discover a simpler but equivalent set of equations for which the theorem could be proved. In practice this means that conditional equations must be substituted with an equivalent set of unconditional ones [Sta87a]. If the inductionless induction procedure does not terminate after this first transformation, further simplification is required. This can be achieved by systematically replacing derived operators in the left hand sides of equations by constructors. It follows that, after each transformation, it must be formally shown that the new set of equations is equivalent to the previous one. This means that every equation in the first set must be shown to be a consequence of the rules in the second set, and every rule in the second set must be shown to be a consequence of the rules in the first set.

4.3 SML and Cambridge LCF

Of the pairs looked at so far, this is the least compatible. Although Standard ML is a descendant of ML which was designed as the metalanguage (in this case a language of proof generating commands) of the LCF system [GMW79], the Standard ML description is of very little use as input to the theorem prover. This is so because the object language of LCF (in which specifications must be expressed) is very different from its metalanguage.

In the tradition of functional languages, SML supports strong typing, polymorphism and higher order functions. The behaviour of the adder is defined as

```
fun adder(c-in,l,nil) =
      if (c-in eq f) then l else inc(l) |
    adder(c-in,nil,l) =
      if (c-in eq f) then l else inc(l) |
    adder(c-in,bit1::rest1,bit2::rest2) =
      sum(fulladder(bit1,bit2,c-in))::
```

```
        adder(carry(fulladder(bit1,bit2,c-in)),
              rest1,rest2) ;
```

where :: is the cons operator and | is choice. Simulation tests are carried out with the by now familiar **test** operator:

```
fun test(n1,n2) =
    bin-to-nat(adder(f,nat-to-bin(n1),nat-to-bin(n2))) ;

test(2213,21) ;
> val it = 2234 : int
```

The theorem prover used here is Cambridge LCF which is a direct descendant of Edinburgh LCF. LCF stands at the opposite end of the spectrum of automatic theorem proving, when compared with REVE and Boyer-Moore. It may more be viewed, more accurately, as an environment for constructing proofs. The logic is $PP\lambda$ which is essentially a typed λ-calculus together with fixed point induction. All sorts have a partial ordering and a bottom element (UU) and inference rules are encoded into *tactics* thus supporting goal-directed (backward) proofs. The partial function principle creates the disadvantage of having to prove the totality of functions before they can be used. Additionally, proofs tend to become cluttered with extra cases due to the bottom element. Proving the adder theorem involves solving the following goal:

```
set_goal([],"!x y c-in. ~x == (UU:(tr)vector) /\
                        ~y == (UU:(tr)vector) /\
                        ~c-in == (UU:tr) ==>
      (bin-to-nat(adder(c-in,x,y))) == bin-to-nat(x) +
                                       bin-to-nat(y) +
                                       bit(c-in)");;
```

where ! is the universal quantifier $\forall$, ~ is the negation predicate and tr is the predefined sort for booleans.

The proof uses the *subgoal package* to break up the original goal into a set of subgoals which are solved individually using inference rules and tactics. The proof trace in this case was 18 pages long. In general, since proof construction is a highly interactive activity, the user must be able to guide the proof. This offers a high degree of confidence in the correctness of the proof, but on the other hand requires a lot of user expertise. In conclusion:

1. in order to make effective use of LCF, the user must be experienced in both proof mechanisms and the system;
2. the very restricted set of pre-proved facts, together with the added complication of dealing with the bottom element cases, can make relatively simple proofs tedious;
3. the proof management facilities (subgoal package), especially in view of the highly interactive nature of the system, are not satisfactory.

The LCF proof of the adder theorem took 3 man weeks. Most of this time was devoted to familiarisation with the system. This fact is indicative of the amount of training required by the users, in order to make effective use of the system.

4.4 SML and HOL

HOL is a version of LCF for higher order logic. It thus shares many features (good and bad) with LCF. SML is still the interface language, but SML descriptions cannot be used as input to the theorem prover. The fundamental problem in this respect is that behaviours in HOL are defined as predicates (relations) and are consequently non executable. In order to simulate such behaviours, it is necessary to translate them into functions. This is not always possible but useful work in this direction is reported in [Cam87].

The logic of HOL is classical higher order logic. The axiomatisation however is different from Church's [Chu40]. The logic is a considerable improvement on LCF in the sense that descriptions are more natural, mainly because of the power of the logic. From the point of view of proof complexity, on the one hand proofs are no longer littered with the bottom element cases, but on the other the power of the logic means that proofs are more complex. As in LCF, the proofs are constructed in a natural deduction-like system. The proposition characterising the behaviour of the adder is expressed in the goal below:

```
set_goal([],"bin-to-nat(adder c-in vector1 vector2) =
             (bin-to-nat vector1) +
             (bin-to-nat vector2) +
             (bit c-in)") ;;
```

Note the absence of the undefinedness checks for the variables.

On the whole, HOL seems to be a definite improvement on LCF. An extra benefit is the reasonable collection of pre-proved facts that comes with the system, thus alleviating the need to prove trivial facts. In spite of this, the proof trace was 35 pages long, mainly because the implementation is largely a prototype with many unrefined features. Particularly problematic areas for this proof are enumerated below.

1. There is no generalised induction package. Thus inductive proofs are only possible on natural numbers. In this instance, inductive arguments over binary vectors were required. Therefore, induction on vectors had to be built up in the logic starting from primitive recursion. This task is difficult and time consuming. Generalised induction must be implemented before HOL can be of real use.
2. A direct consequence of the above problem, is the inability to define recursive functions on anything but natural numbers. Thus, trivially defined functions, such as the length of a vector must again rely on primitive recursion principles.

When these issues are resolved (perhaps in a future implementation), HOL will certainly be a reasonable theorem proving environment choice. In this case, it took 4 man weeks to construct the HOL proof. Almost all the time was spent on understanding and building up generalised induction and the associated procedures from primitive recursion. The lack of suitable induction tactics must also be identified as a source of delay in constructing the proof.

5 Discussion

Formal methods produce the maximal benefit in hardware development when the correct technique is identified and embedded in a traditional framework such as simulation or analysis. A first step toward such integration is the use of formal specifications both as simulation texts and as input to appropriate theorem provers.

We identify the following list of desirable attributes of formalisms intended for use in hardware development:

- simplicity which aids learning, reading and use of a notation;
- expressive power to allow natural descriptions;
- mathematical tractability to reduce the volume of manipulations required for proofs;
- hierarchical structure which supports decomposition;
- processability which affords machine support of the formalism.

We have concentrated on the use of standard functional languages as specification and simulation tools. For the simple example of the n-bit adder, these languages provide as much power in terms of expressiveness and speed as conventional simulators [Sta87b]. It would be a worthwhile exercise to conduct a controlled experiment on the relative performance of these two options.

What became obvious from our theorem proving experiments is that although the area has a firm theoretical basis and useful systems do exist, there must be a compromise of approaches and a combination of techniques, before theorem provers can become an extablished part of the hardware development process. We feel that interactive theorem proving in the flavour of LCF and HOL, enriched with heuristic procedures and augmented with a pre-proved set of facts relevant to hardware systems, is the most promising option. Such a system will also need to have improved user interfaces and proof management facilities.

Although this study concentrated on a simple subset of problems, that is purely combinational devices, it has produced useful insights into various systems and the role of formal techniques in hardware development. We have not, however, investigated behaviour in time and the associated problems. The next phase of the HAVE project will concentrate on just that. Further into the future, the experience gained from this project will be embodied in a hardware designer's workbench supporting formal specification, simulation and formal verification of designs.

Acknowledgements

We would like to thank John Gurd for his continuing enthusiasm, support and encouragement and also Michael Fisher and Peter Lindsay for their helpful comments and suggestions during the preparation of this paper.

References

[Bac78] J. Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bar84] H. G. Barrow. Proving the Correctness of Digital Hardware Designs. *VLSI Design*, 64–77, July 1984.

[BFM85] S. A. Babiker, R. A. Fleming, and R. E. Milne. *A Tutorial for LTS.* Standard Telecommunication Laboratories Ltd, London Road, Harlow, Essex, UK, August 1985.

[BM83] Robert S. Boyer and J. Strother Moore. *Proof-Checking, Theorem-Proving, and Program Verification.* Technical Report 35, Institute for Computing Science and Computer Applications, University of Texas at Austin, January 1983.

[BMS80] R. M. Burstall, D. B. McQueen, and D. T. Sannella. HOPE: An Experimental Applicative Language. In *Procs of LISP Conference*, August 1980.

[Cam87] A. J. Camilleri. *Executing Behavioural Definitions in Higher Order Logic.* Ph.D. dissertation, Cambridge University Computer Laboratory, Corn Exchange Street, Cambridge CB2 3QG, UK, 1987.

[CES83] E. M. Clarke, E. A. Emmerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. In *Procs of 10th ACM Symposium on Principles of Programming Languages*, Austin, Texas, 1983.

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, (5):, 1940.

[Coh87] A. J. Cohn. A Proof of Correctness of the Viper Microprocessor: The First Level. In *Procs of the Calgary Hardware Verification Workshop*, Calgary, Canada, January 1987.

[CP87] P. Camurati and P. Prinetto. Formal Verification of Hardware Correctness: an introduction. In C. J. Koomen and M. Barbacci, editors, *Proc of Computer Hardware Description Languages and their Applications*, Amsterdam, The Netherlands, April 1987.

[CPC87] M. S. Chandrasekhar, J. P. Privitera, and K. W. Conradt. Application of Term Rewriting Techniques to Hardware Design Verification. In *Procs, 24th Design Automation Conference*, Miami, Florida, June 1987.

[ELL86] *The ELLA User Manual.* Praxis Systems plc, 20 Manvers Street, Bath, Avon BA1 1PX, UK, 2.0 edition, 1986.

[FGJM85] K. Futatsugi, J. A. Goguen, J-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Procs of Symposium on Principles of Programming Languages*, pages 52–66, 1985.

[GKB87] J. R. Gurd, C. C. Kirkham, and A. P. W. Bohm. The Manchester Dataflow Computing System. In J. J. Dongarra, editor, *Experimental Parallel Computing Architectures*, pages 177–219, North-Holland, June 1987.

[GMW79] M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* Volume 78 of *Lecture Notes in Computer Science*, Springer Verlag, 1979.

[Gor85] M. J. C. Gordon. *HOL: A Machine Oriented Formulation of Higher Order Logic.* Technical Report 68, Computer Laboratory, University of Cambridge, Corn Exchange Street, Cambridge CB2 3QG, UK, July 1985.

[HD85] F. K. Hanna and N. Daeche. Specification and Verification Using Higher Order Logic. In *Procs of 7th International Symposium on Computer Hardware Description Languages and Applications*, pages 418–443, Tokyo, Japan, August 1985.

[HH82] G. Huet and J. M. Hullot. Proof by Induction in Equational Theories with Constructors. *JCCS*, 2(25):, 1982.

[Hun85] W. A. Hunt. *FM8501: A Verified Microprocessor.* Ph.D. dissertation, University of Texas at Austin, Austin, Texas 78712, December 1985.

[Les83] P. Lescanne. Computer experiments with the REVE Term Rewriting System Generator. In *Proceedings of the Tenth ACM Symposium on the Principles of Programming Languages*, Austin, Texas, Jan 1983.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, edition, 1980.

[Mil83] G. Milne. CIRCAL: A Calculus for Circuit Description. *INTEGRATION, the VLSI Journal*, (1,2 and 3):121–160, 1983.

[Mil84] Robin Milner. Standard ML—The Core Language. July 1984.

[MNP87] D. R. Musser, P. Narendran, and W. J. Premerlani. BIDS: A Method for Specifying and Verifying Bidirectional Hardware Devices. In *Proc of Hardware Verification Workshop*, Calgary, Canada, January 1987.

[Mos83] B. Moszkowski. A Temporal Logic for Multi-Level Reasoning About Hardware. In *Procs of 6th IFIP International Symposium on Computer Hardware Description Languages and their Applications*, Pittsburgh, Pennsylvania, May 1983.

[Pau83] L. Paulson. *The Revised Logic PPLAMBDA: a Reference Manual.* Technical Report 36, University of Cambridge Computer Laboratory, Corn Exchange Street, Cambridge CB2 3QG, UK, 1983.

[Pau85] L. C. Paulson. *Interactive Theorem Proving with Cambridge LCF - A User's Manual.* Technical Report 80, Computer Laboratory, University of Cambridge, Corn Exchange Street, Cambridge CB2 3QG, UK, November 1985.

[PSE85] D. Patel, M. Schlag, and M. Ercegovac. νFP: An Environment for the Multi-level Specification, Analysis, and Synthesis of Hardware Algorithms. In J-P. Jouannaud, editor, *Procs of Conf on Functional Programming Languages and Computer Architecture*, pages 238–255, Springer-Verlag, Nancy, France, September 1985.

[SD84] C. U. Smith and J. A. Dallen. Future Directions for VLSI and Software Engineering. In T. L. Kunii, editor, *VLSI Engineering, Proceedings*, Springer-Verlag, 1984.

[SE87] T. D. Spiers and D. A. Edwards. A High Performance Routing Engine. In *Procs of 24th ACM/IEE Design Automation Conference*, pages 793–799, 1987.

[She84] M. Sheeran. μFP: a Language for VLSI Design. In *Procs of ACM Conference on Lisp and Functional Programming*, pages 104–112, August 1984.

[Sta87a] V. Stavridou. Specifying in OBJ, Verifying in REVE and Some Ideas About Time. October 1987. Department of Computer Science, University of Manchester, Manchester M13 9PL, UK.

[Sta87b] V. Stavridou. *HAVE Project: Phase 1.* , Department of Computer Science, University of Manchester, 1987. forthcoming Technical Report.

[Win86] G. Winskel. Models and Logic of MOS Circuits. In *Proc NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design*, Marktoberdorf, Germany, July 1986.