Why Hardware Verification needs more than Model Checking*

Klaus Schneider[1], Ramayya Kumar[2] and Thomas Kropf[1]

[1] Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, (Prof. D. Schmid), P.O. Box 6980, 76128 Karlsruhe, Germany, e-mail:{schneide|kropf}@ira.uka.de

[2] Forschungszentrum Informatik, Haid-und-Neustraße 10-14, 76131 Karlsruhe, Germany, e-mail:kumar@fzi.de

Abstract. Model checking of temporal propositional logic specifications is a completely automated approach to the verification of digital circuits. One of the main factors that limit the application of such techniques is the size of the problem which can be handled. Many efforts have been undertaken to reduce the space requirements and to speed up the verification algorithms. However, it is shown in this paper, that there are circuits that cannot be specified in model checking approaches in a satisfactory manner, and hence, these circuits cannot be verified by model checking approaches. It is also shown how these circuits can be succinctly specified using higher-order logic, and how they can be verified semi-automatically.

1 Introduction

The aim of hardware verification is to show the absence of design errors in digital circuits by proving certain properties. Properties that are to be verified are specified by the designer, and therefore it is mandatory that specifications should be succinct and easily understandable. For this reason, one of the most important properties of a verification method is the availability of a readable specification language. In the last few years, many different approaches to hardware verification with different specification languages have been presented. All known approaches can be roughly classified into two categories: model based approaches and rule based ones [1].

While model based approaches are mostly based on finite automata or temporal propositional logic, rule based approaches are mostly based on higher order logic. Therefore, rule based approaches have a much more expressive specification language. In contrast to model based approaches, they allow the use of abstraction mechanisms [2] for structure, data and time by using hierarchical descriptions [3], complex data types [4] and special temporal operators [5]. Especially the abstraction from concrete bitwidths is useful for the verification of

* This work has been partly financed by a german national grant, project Automated System Design, SFB No.358.

circuits with large data paths [6]. Once a parameterised circuit is proven to be correct, all real circuits that are simple instantiations of it are automatically correct.

In this paper, it is shown that abstraction mechanisms are not only useful for the reduction of the problem sizes, they are moreover essential for the specification of a plethora of circuits. The main problem for the specification of the presented circuits is that an input that is a boolean tuple has to be interpreted as a natural number and this natural number has to be interpreted as a fixed point of time. The specification of these circuits requires therefore types for time, boolean tuples, natural numbers and functions for 'translating' these types into each other.

In the next section, the basic ideas of abstraction techniques for time and data are presented. In section 3, the example circuits are described and formally specified and the difficulties of their specification with temporal propositional logic are discussed. Section 4 introduces a general proof method based on invariants for generic sequential circuits [5, 7] and indicates how the example circuits can be verified using the invariant approach. The paper concludes with experimental results and conclusions.

2 Abstraction Techniques

2.1 Time Abstraction

Often, the interaction of signals in digital circuits leads to temporal relations which are event-oriented and do not relate to explicit points of time. These event-oriented relations can be described by by temporal operators [8].

For example, if a controller starts a data dependent loop, it is in general unknown, how long this loop will iterate. Therefore we *abstract away* from real time instances and consider only three abstract computation stages of these loops: the start of the loop, the computation phase and the termination of the loop. These 'abstract states' of the circuits can be found in all controllers, irrespective of the number of states that exist in its realisation. In order to reason about such event-oriented state transitions, we define event-oriented temporal operators.

All temporal relations can be expressed by a single temporal operator 'WHEN' on which we will concentrate in the following. Given two higher order expressions b and x of type $\mathbb{N} \rightarrow \mathbb{B}$ and a certain point of time[1] t_0, the expression $[x \;\mathsf{WHEN}\; b]^{(t_0)}$ evaluates to T, if the expressions b and x behave as shown in figure 1, i.e. $[x \;\mathsf{WHEN}\; b]^{(t_0)}$ expresses the fact, that x must hold when the event signal b becomes T the first time after t_0. There are also other operators that can be used as a *base operator* but, in our opinion, the WHEN operator is the most interesting one for the specification of digital circuits. For example, the

[1] Time is modeled by natural numbers, therefore boolean valued signals have the type $\mathbb{N} \rightarrow \mathbb{B}$ and tuples of such signals can be interpreted as complex data type of type $\mathbb{N} \rightarrow \alpha$ with some type $\alpha = \mathbb{B}^n$.

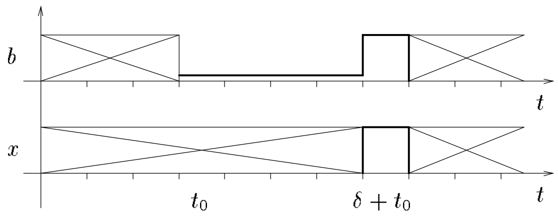

Fig. 1. Timing diagram for the signals satisfying the WHEN operator.

statement – *If a request occurs, then the desired data is valid when acknowledge becomes true for the first time after the request*, is formally stated as follows:

$$\boxed{\forall t.req^{(t)} \rightarrow [data\_valid \ \mathsf{WHEN} \ ack]^{(t)}}$$

There are a several different characterisations of the WHEN operator [5] which can all be used as formal definitions in higher order logic. Each of these characterisations indicates connections to further formalisms, which can thus be embedded in our framework. For example, characterisations of the WHEN operator by hardware formulae, signal descriptions, and invariant characterisation correspond to automata theory, arithmetic and Floyd-Hoare calculi. The characterisation by invariants is especially useful for the verification of generic sequential circuits; we will describe this in section 4.

2.2 Data Abstraction

Digital circuits only work with boolean values, however, it is often the case that some of these boolean values are grouped together to bitvectors. These bitvectors often encode a value of a data type, and the mapping of a bitvector to a value of a complex data type is called data abstraction. In this paper, we will only consider natural numbers as data type.

We use the word library implemented by W. Wong [9] for the specification of the circuits, the data abstraction function BNVAL of this library is however omitted in the following. Instead, we indicate the application of a BNVAL operation by a special font ($\mathfrak{A}$,$\mathfrak{B}$,$\mathfrak{C}$,...), i.e.

$$\mathsf{BNVAL}\underbrace{\left([{b_n}^{(t)}, \ldots, {b_0}^{(t)}]\right)}_{=:\, B^{(t)}} := \underbrace{\sum_{i=0}^{n} \mathsf{BV}\left({b_i}^{(t)}\right) \times 2^i}_{=:\mathfrak{B}^{(t)}}$$

Using the BNVAL operator for the specification of generic circuits requires proofs using more or less difficult theorems of arithmetic. However, it is possible to circumvent this reasoning in the domain of the complex data type by defining corresponding operators directly on the boolean words [4]. These operators have to be validated, i.e. it has to be proven that they correspond with the operator on the complex data type. For example, an increment operation modulo 2^n – SUCC can be defined on boolean words as follows:

$$\begin{array}{l}(\mathsf{SCARRY}([]) = \mathsf{T}) \wedge (\mathsf{SCARRY}(b :: B) := b \wedge \mathsf{SCARRY}(B))\\ (\mathsf{SUCC}([]) := []) \wedge\\ (\mathsf{SUCC}(b :: B) := (\mathsf{SCARRY}(B) \Rightarrow (\neg b) :: \mathsf{SUCC}(B) \mid b :: \mathsf{SUCC}(B)))\end{array}$$

The validation of these operators are the following theorems:

1. $\vdash \forall B.\mathsf{BNVAL}\,(\mathsf{SCARRY}(B)) = 2^{\mathsf{WORDLEN}(B)} - 1$
2. $\vdash \forall B.\mathsf{BNVAL}\,(\mathsf{SUCC}(B)) = \left([\mathsf{BNVAL}\,(B) + 1]\ \mathsf{MOD}\ 2^{\mathsf{WORDLEN}(B)}\right)$

The expansion of the definition of these operators for specific bitwidths leads then to propositional formulae which can be proven by simple tautology checker.

3 Examples

In this section, we specify and verify two examples which cannot be easily handled in temporal propositional logics.

3.1 Summation Circuit

The informal description of this circuit is as follows: the circuit has an n-bit input A, an m-bit input B, and a boolean valued input line req. The outputs of the circuit are an n-bit output OUT and a boolean valued output f. Initially, the circuit is in a 'wait-state' until a new computation is requested (i.e. f is T and req is F). The start of a new computation at time $t_0 + 1$ is initiated by $req^{(t_0)} = \mathsf{T}$ and the inputs $A^{(t_0)}, B^{(t_0)}$ are read. This computation will last for the next $\mathfrak{B}^{(t_0)}$ cycles, and the end of the computation is signaled by the output f. At the end of the computation, the output OUT contains the sum of the last $\mathfrak{B}^{(t_0)}$ values which have appeared on the input A, i.e. $\mathfrak{OUT}^{(t)} = \sum_{j=t_0}^{\mathfrak{B}^{(t_0)}+t_0} \mathfrak{A}^{(j)}$.

A formal specification of the circuit using the temporal operators and the data abstraction operators is as follows:

$$\begin{array}{l}\mathsf{SUMMATION\_SPEC}(n, m, A, B, req, OUT, f) :=\\ \quad \forall t_0.\\ \qquad f^{(t_0)} \wedge req^{(t_0)} \rightarrow\\ \qquad \left[\left(\begin{array}{l}\lambda t.\left(t = \mathfrak{B}^{(t_0)} + t_0 + 1\right) \wedge\\ \quad \left(\mathfrak{OUT}^{(t)} = \left(\left[\sum_{j=t_0}^{t-1} \mathfrak{A}^{(j)}\right]\ \mathsf{MOD}\ 2^n\right)\right)\end{array}\right)\ \mathsf{WHEN}\ f\right]^{(t_0+1)}\end{array}$$

A specification in temporal propositional logic is very complicated since the input B is the computation time of the circuit. A possible specification for a 4 bit instance would be as follows where the sums have to be expressed as a propositional formula ($\bigcirc$ is the nexttime operator):

$$
\begin{aligned}
&B = \text{`}0000\text{'} \Rightarrow [\bigcirc OUT = A] \wedge \\
&B = \text{`}0001\text{'} \Rightarrow [\bigcirc\bigcirc OUT = A + \bigcirc A] \wedge \\
&\vdots \\
&B = \text{`}1111\text{'} \Rightarrow \left[\bigcirc^{16} OUT = A + \bigcirc A + \ldots \bigcirc^{15} A\right]
\end{aligned}
$$

Note that there are 2^m conjuncts, the sums contain up to 2^m summands and the runtime of the model checker is in general exponential in the length of the input formula. An implementation[2] of the circuit is given in figure 2. The controller of the circuit is a simple finite-state machine with two states. The control signals and the transition function of the controller are defined as follows:

- $q^{(t+1)} := req^{(t)} \vee [q^{(t)} \wedge \neg f^{(t)}]$,
- $sel^{(t)} := req^{(t)} \wedge \neg[q^{(t)} \wedge \neg f^{(t)}]$,
- $load^{(t)} := req^{(t)} \vee q^{(t)}$.

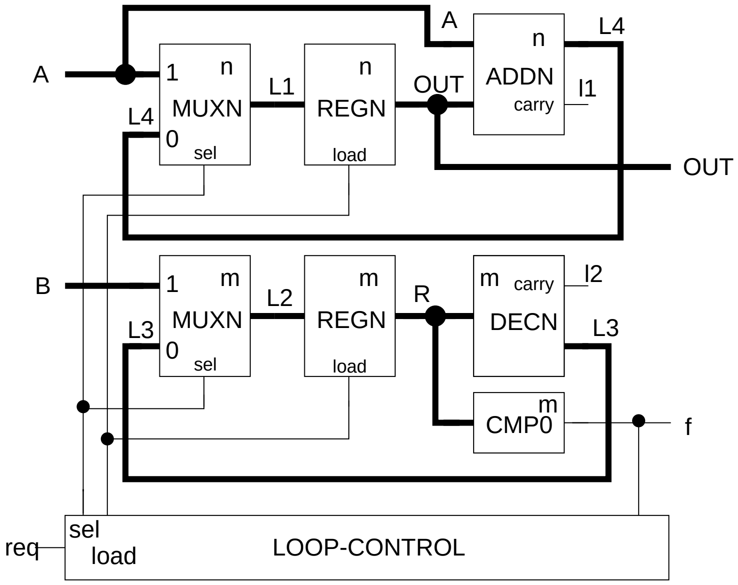

Fig. 2. Implementation of the summation circuit.

3.2 Signal Generator

This is a simple circuit which is nevertheless remarkable as there seems to be no practical way to specify this circuit in temporal propositional logic. Thus it

[2] All components are abstract since they they have either m or n bit broad inputs and outputs.

cannot be verified using model based approaches. The circuit has an n-bit input A and a reset input $reset$ and a boolean valued output c. If $reset^{(t_0)}$ holds for a certain point of time t_0, then the number $A^{(t_0)}$ is loaded into a register and the output c is high. For the next $A^{(t_0)}$ cycles, the output c is low, and for the following cycle it becomes high. This pattern repeats itself infinitely until a new reset occurs[3]. The timing diagram of $reset$ and c is shown in figure 3.

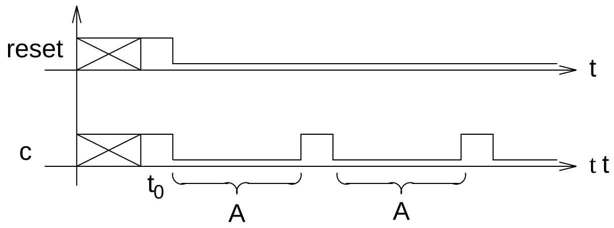

Fig. 3. Timing diagram of the signal generator.

The formal specification of the signal generator is as follows:

$$
\begin{array}{|l|}
\hline
\mathsf{SIGNAL\_SPEC}(n, reset, A, c) := \\
\quad \forall t_0 . reset^{(t_0)} \wedge \left(\forall t . \neg reset^{(t+t_0+1)}\right) \rightarrow \\
\qquad \forall \delta .\ c^{(\delta+t_0)} \rightarrow \left[\left(\lambda t . \left(t = \boldsymbol{A}^{(t_0)} + \delta + t_0 + 1\right)\right) \ \mathsf{WHEN}\ c\right]^{(\delta+t_0+1)} \\
\hline
\end{array}
$$

A specification using model based approaches suffers from the same drawbacks as in the previous example. The implementation of the circuit is straightforward and is given in figure 4. 4.

4 Verification Methods for Generic Sequential Circuits

Concrete circuits can often be verified using decision procedures for the satisfiability of temporal propositional logic, such as model checking. The specifications of the previous section use however higher-order constants such as the BNAL operator. Nevertheless, they can be verified automatically by a model checker if we use validated operators for the addition operation as outlined in section 2. The general case, where we do not instantiate concrete bitwidths can however not be proven this way. In this section, we introduce another proof method which is well suited for the verification of generic sequential circuits as shown in [5].

The higher-order temporal operators have relations to fixed point calculi [10, 11] as the WHEN operator satisfies the following recursion equation:

[3] Periodic behaviour can be described by expressions of the following form: $\forall t . b^{(t)} \rightarrow [x\ \mathsf{WHEN}\ b]^{(t)}$.

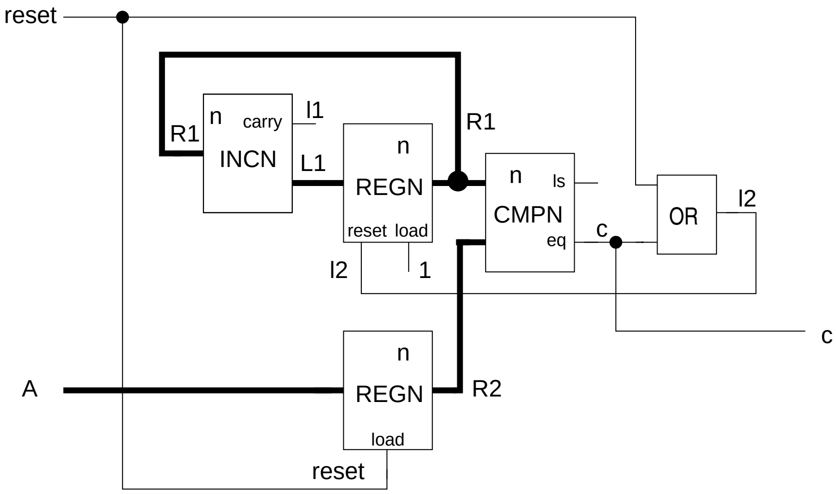

Fig. 4. Implementation of the signal generator.

$$[x \ \mathsf{WHEN}\ b]^{(t_0)} = \left(b^{(t_0)} \Rightarrow x^{(t_0)} \mid \bigcirc [x \ \mathsf{WHEN}\ b]^{(t_0)}\right)$$

In proof procedures for propositional temporal logic, similar recursion equations are used to wrap the temporal operators out until a fixed point is detected. In the higher-order case, however, these iterations need not terminate, thus we cannot establish a decision procedure similar to the propositional case. Instead we propose a more abstract fixed point characterisation using invariants. For this reason, we have proven the following invariant theorem for the WHEN operator:

$$[x \ \mathsf{WHEN}\ b]^{(t_0)} = \left(\begin{array}{l} \exists J. \\ \quad J^{(t_0)} \wedge \\ \quad (\forall t. \neg b^{(t+t_0)} \wedge J^{(t+t_0)} \rightarrow J^{(t+t_0+1)}) \wedge \\ \quad (\forall d. b^{(d+t_0)} \wedge J^{(d+t_0)} \rightarrow x^{(d+t_0)}) \end{array}\right)$$

The bound variable J is called an invariant and the bound subformulae have the following meaning: The first one states that the invariant J holds at the beginning of the computation, the second one states that if the termination condition b does not hold, the invariant implies the invariant at the next time instance, and the third subformula states that if the termination condition b and the invariant J hold at a certain time $\delta + t_0$, then the condition x holds also at time $\delta + t_0$. Invariants are fixed points of iterations, which can be described by the WHEN operator. These subformulae correspond to the subgoals which arise in the verification of WHILE-loops of programming languages in Floyd-Hoare calculi [12] for software verification. The corresponding rule of these calculi is as follows:

$$\frac{\vdash \{P\}\ \mathsf{WHILE}\ B\ \mathsf{DO}\ S\ \{Q\}}{\vdash P \rightarrow J \quad | \vdash \{J \wedge B\} S \{J\} \quad | \vdash \neg B \wedge J \rightarrow Q}$$

Loop invariants express in a formal way the creative design idea of the implementation and are therefore an appropriate means for guiding the verification process. However, if the designer has not given his idea as a loop invariant, an appropriate invariant has to be found afterwards for the verification. In general, loop invariants can however not be found automatically, though heuristics can be used to suggest possible candidates.

The example circuits of the previous section have been verified by using the invariant approach. It has to be noted, that after the invariant rule has been applied with the corresponding invariant, the resulting subgoals are of first-order logic, i.e. they can be proven automatically [13, 14].

For the verification of the summation circuit, the invariant below has been used. This formulae states that during the computation phase, the value of OUT contains the sum of all the values on the input A up to the present time t and additionally, the time elapsed is the difference between B and the value of the internal register R.

$$\begin{array}{l} \lambda t. \\ \quad q^{(t)} \wedge \\ \quad \left(\mathfrak{OUT}^{(t)} = \left(\left[\sum_{j=t_0}^{t-1} \mathcal{A}^{(j)}\right] \mathsf{MOD}\ 2^n\right)\right) \wedge \left(t = \left[\mathcal{B}^{(t_0)} - \mathfrak{R}^{(t)}\right] + t_0 + 1\right) \end{array}$$

The invariant which has been used for the verification of the signal generator is given below. This formula states that the register R_1 corresponds to the current time t and the value of R_1 is less than or equal to R_1 which contains the value of the input A.

$$\lambda t. \left(t = \mathfrak{R}_1^{(t)} + \delta + t_0 + 1\right) \wedge \left(\mathfrak{R}_1^{(t)} \leq \mathfrak{R}_2^{(t)}\right)$$

5 Experimental Results and Conclusions

We have used the presented higher-order temporal operators for the verification of various n-bit circuits with data-dependent loops. The table below gives the runtime and the number of internal inference rules of the correctness proof on a SPARC10 using HOL90. The last column contains the number of states of the n-bit circuit. The first four circuits in the table are different kinds of adders, the others are self-explanatory.

circuit	time [sec]	# Rules	# States
VonNeumann1	7.19	8328	2^{2n}
VonNeumann2	17.33	17578	2^{2n+1}
IncDec-Adder1	6.22	7605	2^{2n}
IncDec-Adder2	15.81	16613	2^{2n+1}
Summation	16.77	20847	2^{n+m+1}
Signal Generator	44.52	81982	2^{2n}
Greatest Common Divisor	27.78	31127	2^{2n+1}
Russian Multiplier	60.14	61777	2^{5n+1}
Divider	73.60	44074	2^{4n+1}

In this paper, it has been shown, that there are relevant circuits which can not be specified in a satisfactory manner using temporal logic as this logic neither allows the use of abstract data types nor an explicit reasoning of time. For this reason, higher-order logic is more and more used for the specification of digital circuits. A verification framework should however encompass both formalisms such that in those cases where higher-order constructs are not required for specifications automatic model checkers can be used.

Previous work has also dealt with the introduction of abstraction mechanisms into model based approaches [15, 16] which allowed the verification of large circuits. These approaches do not enhance the expressive power of the specification language and therefore the circuits presented in this paper can not be verified by them. Therefore we conclude that temporal propositional logic which is the basis for model based approaches is a very important means for the specification and verification of digital circuits, but it can handle only a class of circuits. Therefore model based approaches are not sufficient for hardware verification.

Acknowledgements

The circuits presented in this paper are the result of a discussion with SIEMENS AG, Germany.

References

1. A. Gupta. Formal hardware verification methods: A survey. *Journal of Formal Methods in System Design*, 1:151–238, 1992.
2. T.F. Melham. Abstraction mechanisms for hardware verification. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1988.
3. R. Kumar, K. Schneider, and Th. Kropf. Structuring and automating hardware proofs in a higher-order theorem-proving environment. *Journal of Formal Methods in System Design*, 2(2):165–223, 1993.
4. K. Schneider, R. Kumar, and Th. Kropf. Modelling generic hardware structures by abstract datatypes. In L.J.M. Claesen and M.J.C. Gordon [17], pages 165–176.
5. K. Schneider, T. Kropf, and R. Kumar. Control-path oriented verification of sequential generic circuits with control and data path. In ScKK94a [18].

6. D. Verkest, L. Claesen, and H. DeMan. On the use of the Boyer-Moore theorem prover for correctness proofs of parametrized hardware modules. In L. Claesen, editor, *IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design*, pages 99–116, 1990.
7. K. Schneider, R. Kumar, and Th. Kropf. Automating verification by functional abstraction at the system level. In *International Workshop on Higher-Order Logic Theorem Proving and its Applications*, Malta, 1994.
8. F. Kröger. *Temporal Logic of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1987.
9. W. Wong. Modelling bit vectors in HOL: the `word` library. In *HOL User's Group Workshop*, number 780 in Lecture Notes in Computer Sciences, Vancouver, Canada, August 1993. Springer Verlag.
10. M.Y. Vardi. A temporal fixpoint calculus. In G.J. Milne, editor, *Proc. of the Fifteenth ACM Symposium on the Principles of Programming Languages*, pages 250–269, San Diego, California, 1988.
11. B. Banieqbal and H. Barringer. Temporal logic with fixed points. In *Temporal Logic in Specification*, number 398 in Lecture Notes in Computer Science, pages 62–74, 1987.
12. C.A.R. Hoare. An axiomatic approach to computer programming. *Communications ACM*, 12:576–580, 1969.
13. K. Schneider, R. Kumar, and Th. Kropf. Efficient representation and computation of tableau proofs. In L.J.M. Claesen and M.J.C. Gordon [17], pages 39–58.
14. K. Schneider, R. Kumar, and Th. Kropf. Hardware verification with first-order BDD's. In *Conference on Computer Hardware Description Languages*, 1993.
15. P.A. Subrahmanyam. Towards verifying large(r) systems: A strategy and an experiment. In G.J. Milne and L. Pierre [19], pages 135–154.
16. D.E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie Mellen University, 1993.
17. L.J.M. Claesen and M.J.C. Gordon, editors. *Higher Order Logic Theorem Proving and its Applications*, volume A-20 of *IFIP Transactions*, Leuven, Belgium, 1992. North-Holland.
18. *European Design and Test Conference*. IEEE Computer Society Press, March 1994.
19. G.J. Milne and L. Pierre, editors. *Correct Hardware Design and Verification Methods*, number 683 in Lecture Notes in Computer Science, Arles, France, 1993. IFIP WG10.2 Advanced Research Working Conference, Springer Verlag.