HAL
archives-ouvertes.fr

Symbolic Model Checking with Past and Future Temporal Modalities: Fundamentals and Algorithms

D. Deharbe, D. Borrione

▶ **To cite this version:**

D. Deharbe, D. Borrione. Symbolic Model Checking with Past and Future Temporal Modalities: Fundamentals and Algorithms. Bergé J.M., Levia O., Rouillard J. Model Generation in Electronic Design, springer, pp.105-126, 1995, Model Generation in Electronic Design. <hal-01203695>

HAL Id: hal-01203695

https://hal.archives-ouvertes.fr/hal-01203695

Submitted on 23 Sep 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Symbolic Model Checking with Past and Future Temporal Modalities: Fundamentals and Algorithms

David Déharbe and Dominique Borrione

E-mail: David.Deharbe@imag.fr, Dominique.Borrione@imag.fr
Affiliation: ARTEMIS-IMAG, Université J. Fourier
Address: ARTEMIS, BP 53, 38041 GRENOBLE Cedex 9, FRANCE

Abstract

Model checking is gaining importance in verifying the partial specifications of complex synchronous systems modelled by means of a finite state machine. In this paper, we present the principles and a tool for checking their properties in a temporal logic that allows both past and future oriented modalities. After a revision of the basic concepts of the finite state machine model, and of its representation using binary decision diagrams, we present several algorithms to traverse the set of states symbolically. We then extend CTL with past oriented modalities and give properties of this extended temporal logic (TL). We give algorithms to verify TL formulas by symbolic model checking. A prototype symbolic model checker for TL, taking as input synchronous circuits written in a VHDL subset, has been implemented.

1. Introduction

The only standardised digital systems description language is VHDL, upon which many industrial simulation tools have been developed (see [1,20]). Today, the challenge is to produce highly integrated and completely verified circuits in the shortest time: simulation by exhaustive test sets now cannot take up this challenge. The most recent works on the topic are related to high-level synthesis and formal verification. This paper deals with the second. The goal of formal verification is to establish the correctness of a design with mathematical methods. It should provide the following results: equivalence between two designs, verification of an implementation with respect to a specification, verification of partial correctness (properties).
Many formal proof methods have been studied and developed in order to improve the quality and reliability of circuit designs: symbolic simulation, proof of equivalence, symbolic ternary trajectory evaluation... Among these methods, the proof of properties appears to be of growing importance, as a partial validation method. Assuming the truth of a property consists of making a hypothesis about one aspect of the behaviour of the circuit, and this seems well-adapted to the step by step designer thought process. It is also a valuable method for validating the initial behavioural system specification, given as input to automatic synthesis software. In our approach, we are considering sequential circuits synchronised by a single clock. The semantics of these circuits are defined in terms of a finite state machine model: the memory elements of the circuit constitute the state variables of the finite state machine, and the 'state space' is the combination of values that these

memories can hold. Temporal properties can be expressed in different temporal logics (TL for short) ([17,26]), with different expressive powers: LTL, CTL, CTL*, etc. For instance, CTL is a propositional, branching time TL, it can express both *liveness* (invariants that must be always true in the system) and *safety* (requirements that need not be always true but which achievement have to be always realisable) properties. The common aspect of these logics on which model checking tools are based, is that they are 'future'-oriented: their temporal operators only refer to the future behavior of the system being specified. We propose to come back to the classical TLs, with operators referring to both past and future. Our approach is based on a CTL-like TL. The proof principle is the following: given the model, and the properties to check, the model checking is done by the traversal of the model state space. Another solution is possible, that has been chosen within, for instance, the Lustre environment [3,30,33]: to build a new machine that is equivalent to the formula and merge both machines to produce a product machine with a single output. The proof of the property is then equivalent to the computation of the output on the set of reachable states.
The next step has been to find a compact data representation to compute proofs on larger circuits than the extensive representation (actual enumeration of the states) is able to process. This became possible with binary decision diagrams, called BDD's, a powerful representation of boolean expressions, invented by Akers [2] and Bryant [5]. The BDD's were first applied to the proof of equivalence of combinatorial and synchronous sequential circuits. The representation of boolean formulas with BDD's has been the object of intensive research in order to improve the efficiency of their manipulation in space and computation time [6,9]. BDD's have also been proposed as a more efficient representation of finite state machines, in relation to state space algorithms [7,14,34]. That research resulted in the development of *symbolic* model checkers, where sets of states are kept implicit.
This paper is organized as follows. Section 2 presents the finite state machine model and the principles of state space traversal using binary decision diagrams. Section 3 defines the temporal logic with future and past oriented modalities, and gives a set of normalization and simplification rules on temporal logic formulas. Section 4 gives the BDD-based algorithms that constitute the core of the model checker. Section 5 shows the application of model checking on an example described in VHDL, and presents the prototype software that we have implemented. Section 6 states our conclusions.

2. Fundamentals

An algorithm which verifies that a given formula is true in a given model is called a model checker. In our case, the formulas are TL expressions, these formulas are checked on a finite state machine representing a synchronous sequential circuit. The finite state machine is binary encoded, and therefore the definition is given in terms of boolean constructs.

2.1 The finite state machine model

2.1.1 Definitions

The finite state machine is modelled by means of atomic propositions, so that it is possible to process it with boolean operations ($B = \{True, False\}$ denotes the usual boolean domain). Bose and Fisher [4] defined a model $M = (S, I, O, \delta, \lambda, \Sigma_I)$ of a deterministic finite state machine, as follows:

- S is a power of B, that represents the states of the machine, $S=B^{ns}$, and $s_1, s_2, ..., s_{ns}$ are the corresponding state variables.
- I is a power of B, that represents the inputs of the machine, $I = B^{ni}$, and $i_1, i_2, ..., i_{ni}$ are the corresponding input variables.
- O is a power of B, that represents the outputs of the machine, $O=B^{no}$, and $o_1, o_2, ..., o_{no}$ are the corresponding output variables.
- $\delta: S \times I \rightarrow S : \delta= [\delta_0, \delta_1, ..., \delta_{ns}]$, δ represents the next state function, and $\delta_i: S \times I \rightarrow B$ is the transition function of the state variable s_i.
- $\lambda: S \times I \rightarrow O$. $\lambda= [\lambda_0, \lambda_1, ..., \lambda_{no}]$, λ represents the output function, and $\lambda_i: S \times I \rightarrow B$ is the output function associated to the variable o_i.
- $\Sigma_I \in S$, represents the initial state of the machine.

Definition 2.1 A machine state *is represented by a unique valuation s of the state variables of the model.*

Any state of the machine is binary encoded into some valuation of the state variables of the model. The encoding from the set of machine states to the set of valuations is injective: two states are equal if they are represented by the same valuation.

Definition 2.2 A machine configuration *is represented by a unique valuation* $x = (s,i,o)$ *of the variables of the model, such that* $o=\lambda(s,i)$.

A configuration $x=(s,i,o)$ is said to be *associated* to the state represented by s. When no ambiguity exists, we shall denote Σ_I an arbitrary configuration associated to the initial state Σ_I.

Definition 2.3 *A machine state s' is said to be a* successor *of a machine state s if and only if* $\exists\, i \in I.\ s'=\delta(s,i)$.

As an extension, a machine configuration $x'= (s',i',o')$ is said to be a *successor* of a machine configuration $x = (s, i, o)$ if and only if $s'=\delta(s,i)$.
We shall denote this relation with the predicate *Succ*, both for machine states and configurations: $Succ(s,s')$ and $Succ(x,x')$.

Definition 2.4 *A* state path *is a possibly infinite sequence of machine states, denoted* $(s_0,s_1...s_n,...)$ *such that* $Succ(s_i,s_{i+1})$. *For a finite path* $(s_0,s_1,...s_n)$, *n is the* length *of the path.*
A configuration path *is a possibly infinite sequence of machine configurations, denoted* $(x_0,x_1,...x_n,...)$ *such that* $Succ(x_i,x_{i+1})$. *For a finite path* $(x_0,x_1,...x_n)$, *n is the* length *of the path.*

When no ambiguity exists, we shall simply talk about a path.

Definition 2.5 *A machine state is reachable if there is a finite state path* $(s_0,s_1,...s)$ *such that* $s_0 = \Sigma_I$. *Similarly, a machine configuration x is reachable if there is a finite configuration path* $(x_0,x_1,...x_n)$, *such that* $x_0 = \Sigma_I$.

2.1.2 The reachable configurations of the model

Let X be the set of reachable machine configurations, i.e. the set of machine configurations found on all paths starting from Σ_I. For any natural integer $n \in N$, X_n is the set of machine configurations found on all paths starting from Σ_I of length lower or equal to n.. It is defined by:

- $X_0 = \{ \Sigma_I \}$
- $X_{n+1} = X_n \cup \{ x \mid \exists x '.(x' \in X_n \wedge Succ(x, x'))\}$

The machine has a finite number of configurations. This implies that there exists a k such that $X_k = X_{k+1}$: X_k is the set of reachable machine configurations. Thus, our algorithm to compute the set X just calculates iteratively the limit of $(X_n)_{n \in N}$. X is the fixed point of the condition K in the expression:

$$K = \{x \mid \Sigma_I \vee x \in K \vee \exists x '.(x' \in K \wedge Succ(x, x'))\}$$

The set of reachable states of a finite state machine is defined in the same way.

2.2 Representation of boolean expressions with BDD's

Binary decision diagrams (BDD's) are a representation of boolean expressions, more compact than traditional conjunctive or disjunctive forms. A recent survey on BDD's can be found in [6]. They are direct acyclic graphs representing binary decision trees, where common subtrees have been merged. (Fig.1 shows an example of BDD). Bryant proved in [5] that this representation is canonical for a selected order on the variables.

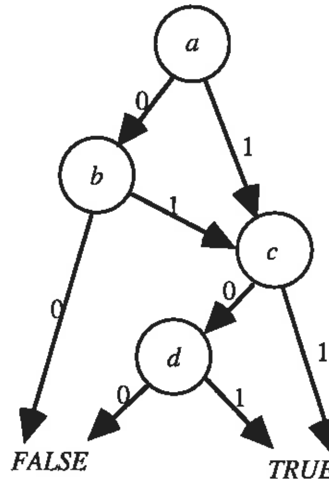

Figure 1: BDD of the boolean expression $(a \vee b) \wedge (c \vee d)$ with $a < b < c < d$

The size of the BDD representing an expression is closely related to the chosen variable ordering. The computation of the best order being NP-hard, heuristics are used to choose this order [9].

2.2.1 Definition

BDD's are based on Boole's decomposition of boolean expressions [5]. Given a finite set of boolean variables $S=\{x_1,..., x_n\}$, a strict order on this set ($x_i < x_j$ if and only if $i<j$) , the BDD representing a boolean expression $E(x_1,..., x_n)$ over this set of variables is inductively defined by:

- if $E(x_i,..., x_n)= True$ or $E(x_i,..., x_n)= False$, the BDD is a single terminal node representing *True* or *False* .
- if $E(x_i,..., x_n) \neq True$ and $E(x_i,..., x_n) \neq False$, the BDD is the graph constructed from one node representing variable x_i , from which two edges depart, one leading to the BDD representing $E(True,x_{i+1},..., x_n) = F(x_{i+1},..., x_n)$, the other leading to the BDD representing $E(False,x_{i+1},..., x_n) = G(x_{i+1},..., x_n)$.

Another improvement, proposed in [25], is to type BDD edges by a sign '+' or '-', depending on whether the value of the expression represented by the adjacent BDD has to be negated or not. The negation complexity is $O(1)$, and the conjunction and the disjunction are $O(n)$, where n is the number of nodes of the manipulated BDD's. But it has been proved that, whatever improvement is made on BDD's, boolean expressions (e.g. the multiplication) can be exhibited such that their BDD representation has an exponential number of nodes. Nevertheless, experience shows that, in general, this representation gives interesting results in terms of operations on boolean expressions; and it is increasingly used for building formal verification tools.

2.2.2 Representation of a finite state machine with BDD's

We are interested in verifying properties (safety, liveness, etc.) over the states of a finite state machine. These properties are expressed in a branching time logic, called TL, which will be defined in section 3. Given a finite state machine model $M=(S, I, O, \delta, \lambda, \Sigma_I)$ and a TL formula f on this machine, our goal is to find the set of reachable states where f is valid. Finite state machines can be efficiently represented with BDD's [4,8,14]. The precise representation we selected is the same as [4] :

- state, input and output variables, elements of
  $$\{s_j, j\in[0,ns]\}\cup\{ i_j, j\in[0,ni]\}\cup\{ o_j, j\in[0,no]\}$$
  are boolean variables, each one is a BDD representing the boolean expression reduced to a single proposition.
- elements of δ and λ are boolean combinations on elements of
  $$\{s_j, j\in[0,ns]\}\cup \{i_j, j\in[0,ni]\},$$
  they are represented by the BDD corresponding to their expression.
- Σ_I is the initial valuation of the elements of $\{s_j, j\in[0,ns]\}$, it is the BDD representing the conjunction of the initial values.

It is possible to represent any machine configuration (e.g. Σ_I, the initial configuration) or set of machine configurations by means of BDD's:

- a machine configuration is the BDD representing the conjunction of the valuations of its elements.
- a set of machine configurations is the BDD representing the characteristic function of this set, that means the disjunction of the BDD's representing the machine configurations of this set.

In conclusion, BDD's are an elegant and efficient solution to manipulate sets of states of a finite state machine. This technique is called *symbolic*, for it manipulates sets of states instead of the states themselves.

2.2.3 Operations on BDD's

The algorithm of the symbolic model checker involves some non-trivial manipulations of boolean formulas: quantification of an expression by a variable, substitution in an expression of the instances of a variable by an expression. These operations are defined in terms of the basic boolean functions as follows (remark: $f|_{x=True}$ and $f|_{x=False}$ are the two sub-graphs of the BDD that represents f, if x is the variable associated to the root node of this BDD):

- existential quantification: $\exists x.f = f|_{x=True} \vee f|_{x=False}$
- universal quantification: $\forall x.f = f|_{x=True} \wedge f|_{x=False}$
- substitution: $f<x \leftarrow v> = \exists x.((x \Leftrightarrow v) \wedge f)$.

We had a very efficient BDD package written by Höreth [19] at our disposal. It has several programming and theoretical improvements, such as hash tables, a memory manager with a garbage collector, various ordering heuristics. It holds the basic boolean expressions: not, or, and, nor, nand, xor, equivalence, implication, but also the more complex operations described above.

2.2.4 Traversal of the symbolic model

The traversal of a finite state machine M from a set of states S consists of computing the image (forward traversal) or the inverse image (backward traversal) of this set by the transition function. Two application examples of machine traversal are the computation of the set of reachable states from Σ_I and of the set of states where a TL formula is valid.
Most of the algorithms to perform the symbolic traversal of a Mealy machine were published by Berthet, Coudert, Madre (see [12,13,14,25]), Burch, Clarke *et al.* (see [7,8]). Their respective works differ in the way the transition function is represented:

- Madre *et al.* use a vector of $ns+no$ functions of $ns+ni$ boolean variables, one for each different internal variable and output, this vector of functions represents both δ and λ.
- Burch *et al.* use a single transition relation, which is a boolean expression of $ni+2(ns+no)$ variables, built with a conjunction of the δ_i and λ_i.

Traversal algorithms based on the transition vector appear to be much more efficient than the ones based on the transition relation method. Some recent works [10] on transition relation-based methods have filled the performance gap between these two approaches.

3. The temporal logic

3.1 Introduction

Modal logics allow one to reason in different modes of logic. For instance, TL (Temporal Logics) are used to study the evolution of the truth of formulas as time evolves. Pnueli showed the usefulness of TL to check the validity or to specify the behavior of programs or

reactive systems [28]. Our approach differs from Pnueli's in the sense that he considers properties of a system in the initial state whereas we define the validity of formulas globally, i.e. in each state of the system: in this sense, the solution of a formula in a system (in our context, a finite state machine), is the set of states where the formula is valid. This approach is the same as the one usually used to define the validity of Computation Tree Logic — CTL — formulas (see e.g. [11,29]) or fragments of the μ -calculus [8]. We show that, in this sense, the expressive power of a full logic is strictly bigger than the future-oriented logics.

3.2 Definition

As a basis, we take the logic defined in [26] which is a full-time logic. In Emerson's framework of classification, our TL has the following attributes: it is propositional, global, branching, evaluated on points, discrete and has both past-tense and future-tense operators. It is presented in a CTL-like notation, as it is a straightforward extension of CTL. CTL operators define properties on the future behavior. New operators are added so that the same expressiveness is now possible on the past behavior. Similar operators have also been defined on linear TL, in [17].

Informally, one defines TL formulas as expressions built upon boolean propositions, i.e. the set $P = \{ s_j, j \in [0,ns]\} \cup \{ i_j, j \in [0,ni]\} \cup \{ o_j, j \in [0,no]\}$, with the classic boolean connectives (negation, conjunction, etc.) and the TL connectives. A TL connective is a quantifier on paths (A: all, E: exist) followed by a modality over the states of a path (*X*: next time, *P*: previous time, *F*: sometimes in the future, *B*: sometimes in the past, *G*: always in the future, *H*: always in the past, *U*: until, *S*: since).
Let R be the set of reachable machine configurations, x, x' be some machine configurations, $p \in P$ a boolean proposition, f, f_1, f_2 TL formulas over P ; the validity of TL formulas is inductively defined by the following:

- $x < p$, where $p \in P$ iff the valuation of p in x is 1 . In the machine configuration x, the boolean proposition p is *True* .
- $x < \neg f =_{def}$ not $x < f$.
- $x < f_1 \wedge f_2 =_{def} x < f_1$ and $x < f_2$. The boolean operators have their usual meaning.
- $x < EXf =_{def} \exists x' = (s', i', o').\ (\delta(s, i) = s' \wedge x' < f)$. EXf is valid in x if there exists a successor of x where f is valid.
- $x < AXf =_{def} \forall x' = (s', i', o').\ (\delta(s, i) = s' \wedge x' < f)$. AXf is valid in x if f is valid in every successor of x.
- $x < EPf =_{def} \exists x' = (s', i', o').\ (\delta(s', i') = s \wedge (x' \in R \Rightarrow x' < f))$. EPf is valid in x if there exists a reachable predecessor of x where f is valid.
- $x < APf =_{def} \forall x' = (s', i', o').\ (\delta(s', i') = s \wedge (x' \in R \Rightarrow x' < f))$. APf is valid in x if f is valid in every reachable predecessor of x.
- $x < EFf =_{def} \exists (x_0, x_1, \ldots).\ (x = x_0 \wedge \exists j \geq 0 .\ (x_j < f))$. EFf is valid in x if there exists a path starting from x, such that there is a state on this path where f is valid. f is *possible* in the future.
- $x < AFf =_{def} \forall (x_0, x_1, \ldots).\ (x = x_0 \wedge \exists j \geq 0 .\ (x_j < f))$. AFf is valid in x if for every path starting from x, there is a state on this path where f is valid. f is *unavoidable* in the future.

- $x < EBf =_{def} \exists(x_0, x_1, ..., x_n).\ (x_0=\Sigma_I \wedge x_n=x \wedge \exists j\ .\ (0 \leq j \leq n) \wedge (x_j < f))$. EBf is valid in x if there exists a path from the initial state ending in x, such that f is valid in some state on this path. f happened *possibly once*.
- $x < ABf =_{def} \forall(x_0, x_1, ..., x_n).\ (x_0=\Sigma_I \wedge x_n=x \wedge \exists j\ .\ (0 \leq j \leq n) \wedge (x_j < f))$. ABf is valid in x if for every path from the initial state ending in x, there is some state on this path where f is valid. f happened *at least once*.
- $x < EGf =_{def} \exists(x_0, x_1, ...).\ (x=x_0 \wedge \forall j \geq 0\ .\ (x_j < f))$. EGf is valid in x if there is a forward path starting from x on which f *globally holds*.
- $x < AGf =_{def} \forall(x_0, x_1, ...).\ (x=x_0 \wedge \forall j \geq 0.(x_j < f))$. AGf is valid in x if for each forward path from x, f globally holds, f is *always valid in the future*.
- $x < EHf =_{def} \exists(x_0, x_1, ..., x_n).\ (x_0=\Sigma_I \wedge x_n=x \wedge \forall j\ .\ (0 \leq j < n) \wedge (x_j < f))$. EHf is valid in x if there is a path from the initial state to x on which f globally holds.
- $x < AHf =_{def} \forall(x_0, x_1, ...x_n).\ (x_0=\Sigma_I \wedge x_n=x \wedge \forall j\ .\ (0 \leq j < n) \wedge (x_j < f))$. AHf is valid in x if for every path from the initial state to x on which f globally holds, f is *always valid in the past.*
- $x < E[fUg] =_{def} \exists(x_0, x_1, ...).\ (x=x_0 \wedge \exists n \geq 0.\ (x_n < g \wedge \forall k\ .\ (0 \leq k < n \Rightarrow (x_k < f)))$. $E[fUg]$ is valid in x if there is a forward path from x , where f is valid on every configuration *until* a configuration where g is valid is met.
- $x < A[fUg] =_{def} \forall(x_0, x_1, ...).\ (x=x_0 \wedge \exists n \geq 0.\ (x_n < g \wedge \forall k\ .\ (0 \leq k < n \Rightarrow (x_k < f)))$. $A[fUg]$ is valid in x if on every path starting from x, f is valid on every configuration *until* a configuration where g is valid is reached.
- $x < E[fSg] =_{def} \exists(x_0, x_1, ...x_n).\ (x_0=\Sigma_I \wedge x_n=x \wedge \exists j \leq n\ .\ (x_j < g \wedge \forall k\ .\ (j < k \leq n) \Rightarrow (x_k < f)))$. $E[fSg]$ is valid in x if there is a path ending to x, where f is valid on every configuration *since* a configuration where g was valid has been left.
- $x < A[fSg] =_{def} \forall(x_0, x_1, ...x_n).\ (x_0=\Sigma_I \wedge x_n=x \wedge \exists j \leq n\ .\ (x_j < g \wedge \forall k\ .\ (j < k \leq n) \Rightarrow (x_k < f)))$. $A[fSg]$ is valid in x if on every path ending to x, f is valid on every configuration *since* a configuration where g was valid has been left.

Remarks

There is a duality in the meaning of the modalities X and P, F and B, G and H, U and S. Nevertheless, their respective definitions are not completely symmetrical. This is the consequence of the non-stability of the set of reachable states with respect to the inverse image function. Actually: $(s' \in R) \wedge (\delta(s,i) = s') \pm s \in R$.

We also had to make a choice on the paths considered when looking to the past: actually, in an automaton, when considering a reachable state, and going infinitely back in the past, it might be possible not to meet the initial state. A typical example is a sink state, which, as long as it has itself as a successor, has therefore also itself as a predecessor. Thus, starting from this sink state, there is an infinite path that goes back in the past and remains in this state. Obviously, if it is not the initial state, this sequence should not be considered. So, it is important to notice that a restriction has to be done on paths when looking to the past. In our algorithms, only paths starting from the initial state are considered.

3.3 Properties of TL

Definition 2.1 *Two TL formulas f and g are equivalent, and denoted $f \cong g$, if :*

$$\forall x \in R \,.\, (x \prec f \equiv x \prec g),$$

where R denotes the set of machine configurations.

This definition of equivalence between TL formulas is different from the one usually used [24,28], which is an equivalence based on the value of the formulas in the initial state only. We claim that our definition is more relevant in the scope of model checking, for two reasons:

1. The semantics of the TL formulas are given for each state of the FSM.
2. Our algorithms are able to determine the validity of formulas in each state of the FSM's (these algorithms are given below in section 4).

This definition of equivalence allows us to define some properties on TL formulas. The first set of properties, the *normalization* rules, are used to get normal forms of TL formulas. The second set of properties are *simplification* rules in the sense that they allow to get equivalent but simpler TL formulas: they decrease the number of TL operators.

The normalization rules

From the definitions of the operators of TL, it is obvious to prove the following equivalencies in TL, where f and g are TL formulas:

(1) $AXf \cong \neg EX\neg f$ (2) $APf \cong \neg EP\neg f$
(3) $EFf \cong E[True U f]$ (4) $EBf \cong E[True S f]$
(5) $AFf \cong \neg EG\neg f$ (6) $ABf \cong \neg EH\neg f$
(7) $AGf \cong \neg EF\neg f$ (8) $AHf \cong \neg EB\neg f$
(9) $A[fUg] \cong \neg E[\neg g U(\neg f \wedge \neg g)] \wedge \neg EG\neg g$ (10) $A[fSg] \cong \neg E[\neg g S(\neg f \wedge \neg g)] \wedge \neg EH\neg g$

If we use these equations as left to right rewrite rules, the following proposition holds:

Proposition 3.1 *Any TL formula can be rewritten in terms of the six TL operators EX, EP, EG, EH, EU, ES.*

Simplification rules

If we restrict ourselves to the future fragment of TL, we get all the properties of CTL. These properties are used to simplify the formulas to be proved. The formulas we present are taken from [18]. In addition, we present the dual property in the past fragment of TL. Let f, f', g, g' be TL formulas:

(11) $AG(f) \wedge AG(g) \cong AG(f \wedge g)$ (12) $AH(f) \wedge AH(g) \cong AH(f \wedge g)$
(13) $EF(f) \vee EF(g) \cong EF(f \vee g)$ (14) $EB(f) \vee EB(g) \cong EB(f \vee g)$
(15) $AG(AG(f)) \cong AG(f)$ (16) $AH(AH(f)) \cong AH(f)$
(17) $EF(EF(f)) \cong EF(f)$ (18) $EB(EB(f)) \cong EB(f)$
(19) $AG(EF(AG(EF(f)))) \cong EF(AG(EF(f)$ (20) $AH(EB(AH(EB(f)))) \cong EB(AH(EB(f)))$
(21) $EF(AG(EF(AG(f)))) \cong AG(EF(AG(f)))$ (22) $EB(AH(EB(AH(f)))) \cong AH(EB(AH(f)))$
(23) $EG(EG(f)) \cong EG(f)$ (24) $EH(EH(f)) \cong EH(f)$
(25) $AF(AF(f)) \cong AF(f)$ (26) $AB(AB(f)) \cong AB(f)$
(27) $AF(EG(AF(f))) \cong EG(AF(f))$ (28) $AB(EH(AB(f))) \cong EH(AB(f))$
(29) $EG(AF(EG(f))) \cong AF(EG(f))$ (30) $EH(AB(EH(f))) \cong AB(EH(f))$
(31) $AG(EG(f)) \cong AG(f)$ (32) $AH(EH(f)) \cong AH(f)$
(33) $EF(AF(f)) \cong EF(f)$ (34) $EB(AB(f)) \cong EB(f)$

(35) $AF(AG(AF(f))) \cong AG(AF(f))$ (36) $AB(AH(AB(f))) \cong AH(AB(f))$
(37) $AG(AF(AG(f))) \cong AF(AG(f))$ (38) $AH(AB(AH(f))) \cong AB(AH(f))$
(39) $A[fUA[fUg]] \cong A[fUg]$ (40) $A[fSA[fSg]] \cong A[fSg]$
(41) $EG((EG(f) \vee g) \vee (f \vee EG(g))) \cong EG(f) \vee EG(g)$
(42) $EH((EH(f) \vee g) \vee (f \vee EH(g))) \cong EH(f) \vee EH(g)$
(43) $AF((f \wedge AF(g)) \vee (AF(f) \wedge g)) \cong AF(f) \wedge AF(g)$
(44) $AB((f \wedge AB(g)) \vee (AB(f) \wedge g)) \cong AB(f) \wedge AB(g)$
(45) $AF((f \wedge f')\ U\ ((g \wedge A[f'Ug']) \vee (g' \wedge A[fUg]))] \cong A[fUg] \wedge E[f'Ug']$
(46) $A[(f \wedge f')\ S\ ((g \wedge A[f'Sg']) \vee (g' \wedge A[fSg]))] \cong A[fSg] \wedge A[f'Sg']$

Proposition 3.2 *The rewrite system made of rules 11 to 46 is nœtherian (i.e. terminates).*

Proof: Let us associate to a TL formula f a measure $M(f)$ equal to the number of TL connectives that f contains. Every simplification rule applied to f makes $M(f)$ decrease. Thus the rewrite system terminates.

Proposition 3.3 *The rewrite system made of rules 11 to 46 is not confluent.*

Proof: By counter-example. Let $f = EF(EF(g)) \vee EF(h)$.
Applying rule 13: $f \rightarrow EF(EF(g)) \vee h)$ and the rewriting stops.
Applying rule 17 and 13: $f \rightarrow EF(g) \vee EF(h) \rightarrow EF(g \vee h)$ and the rewriting stops.
The system is terminating but is not locally confluent, thus according to the Diamond Lemma [27], it is not confluent.

Proposition 3.4 *The rewrite system made of rules 15 to 40 is confluent.*

Proposition 3.5 *The rewrite system made of rules 11 to 14 and 41 to 46 is confluent.*

Proof: The system has been proved to be complete, using the automated tool REVE [23]. Four new simplification rules have been found. The whole REVE session is reported in [16].

Algorithm: We propose the following strategy :

1. Apply iteratively the following steps until the rewriting stops :
   (a) apply rewriting system made of rules 15 to 40,
   (b) apply rewriting system made of rules 11 to 15 and 41 to 46.
2. Normalization with the rules 1 to 10.

Expressive power
Emerson proved in [17] that the propositional linear temporal logic defined globally (see definition 2.1) is more expressive with both past and future operators than the one with future operators only.

Proposition 3.6 *As measured to global equivalence, TL is strictly more expressive than CTL.*

Proof[1] **:** Let M and M' be two finite state machines, completely identical, except their initial state, equal respectively to Σ and Σ'. Moreover, assume $\delta(i,\Sigma)=\delta(i,\Sigma')$, for all inputs i. TL is able to distinguish the two structures: $AP(\Sigma)$ is different for the two machines, whereas CTL cannot make a difference between immediate successors of the initial states, since the paths starting from these states are identical.

4. Algorithms of the symbolic model checker

Given a finite state machine M and a formula F, the model checker returns the characteristic function of the set of reachable configurations of M where F is valid. The algorithm first recurses over the structure of the TL formula F to be proved, with a recursive function *BddF* that returns the BDD corresponding to the characteristic function of the set of machine configurations where the formula is valid. When this first step is achieved, the intersection (i.e. a conjunction) of this first result and of the characteristic function of reachable machine configurations is performed, and it produces the expected result.

Let *InverseImage* and *Image* be two functions that, given a FSM M, and a BDD B that represents a set of configurations, respectively return the inverse image and the image of B by the transition function of M.
The algorithm in each case is:

- F is a pure boolean formula, *BddF* is simply the BDD corresponding to F.

- $F = EXg$: *BddF* is the BDD of the characteristic function of the set
  $$S=\{ x : \exists x' (x' \in Bddg.\ x'=Succ(x)) \},$$
  so *BddF=InverseImage (M, Bddg) and M. BddR*[2] .

- $F = EPg$: *BddF* is the BDD of the characteristic function of the set
  $$S=\{ x : \exists x' (x' \in Bddg \text{ and } Succ(x')=x) \},$$
  so *BddF=Image (M, Bddg and M. BddR)* .

- $F = EGg$: *BddF* is the fixed point of the condition K in the expression:
  $$K=g \wedge EX(K)$$
  BddF is the result of the iterative function CHECK_EG_FORMULA below.

```
function CHECK_EG_FORMULA (M: Symbolic_Model, Bddg: BDD) : BDD;
    -- The function returns the BDD that represents the set of
    -- valuations where EG(g) is valid.
  var
    SOLUTION, INV_SOLUTION: BDD;
    -- INV_SOLUTION represents the inverse image of SOLUTION
  begin
    SOLUTION := Bddg;
```

---

[1] This proof is very similar to the one given by Emerson for Linear Temporal Logic [17].

[2] *M.BddR* and *M.BddInit* are respectively the BDD's of the set of reachable states and of the initial state of *M*.

```
        repeat
           INV_SOLUTION := InverseImage(M, SOLUTION);
           SOLUTION := SOLUTION and (Bddg and INV_SOLUTION);
        until (SOLUTION = INV_SOLUTION);
        return(SOLUTION);
    end CHECK_EG_FORMULA;
```

- $F = EHg$: *BddF* is the fixed point of the condition K in the expression:

$$K = g \wedge EP(K)$$

BddF is the result of the iterative function CHECK_EH_FORMULA below.

```
function CHECK_EH_FORMULA (M: Symbolic_Model, Bddg: BDD) : BDD;
        -- The function returns the BDD that represents the set of
        -- reachable valuations where EH(g) is valid.
    var
        SOLUTION, IMG_SOLUTION: BDD;
        -- IMG_SOLUTION represents the image set of SOLUTION
begin
    SOLUTION := Bddg and M.BddInit;
    repeat
        IMG_SOLUTION := Image(M, SOLUTION);
        SOLUTION := SOLUTION and (Bddg and IMG_SOLUTION);
    until (SOLUTION = IMG_SOLUTION);
    return(SOLUTION);
end CHECK_EH_FORMULA;
```

- $F = E[gUh]$: *BddF* is fixed point of the condition K in the expression:

$$K=h \vee (g \wedge EX(K))$$

BddF is the result of the iterative function CHECK_EU_FORMULA below.

```
function CHECK_EU_FORMULA (M: Symbolic_Model, Bddg, Bddh: BDD) : BDD;
        -- The function returns the BDD that represents the set of
        -- reachable valuations where E(gUh) is valid.
    var
        SOLUTION, INV_SOLUTION: BDD;
        -- INV_SOLUTION represents the inverse image set of SOLUTION
begin
    SOLUTION := Bddh;
    repeat
        INV_SOLUTION := InverseImage(M, SOLUTION);
        SOLUTION := Bddh or (Bddg and INV_SOLUTION);
    until (SOLUTION = INV_SOLUTION);
    return(SOLUTION);
end CHECK_EU_FORMULA;
```

- $F = E[gSh]$: *BddF* is fixed point of the condition K in the expression:

$$K=h \vee (g \wedge EP(K))$$

BddF is the result of the iterative function CHECK_ES_FORMULA below.

```
function CHECK_ES_FORMULA (M: Symbolic_Model, Bddg, Bddh: BDD) : BDD;
        -- The function returns the BDD that represents the set of
        -- reachable valuations where E(gSh) is valid.
```

```
    var
        SOLUTION, IMG_SOLUTION: BDD;
        -- IMG_SOLUTION represents the inverse image set of SOLUTION
begin
    SOLUTION := Bddh and M.BddR;
    repeat
        IMG_SOLUTION := Image(M, SOLUTION;
        SOLUTION := Bddh or (Bddg and IMG_SOLUTION);
    until (SOLUTION = IMG_SOLUTION);
    return(SOLUTION);
end CHECK_ES_FORMULA;
```

As we previously stated, any TL operator can be rewritten in terms of the six operators *EX*, *EP*, *EF*, *EH*, *EU*, *ES*. This is the method we employed in our model checker. Given the normalized TL formula, the algorithm simply recurses over the structure of this formula and applies the six basic algorithms according to the current TL operator.

5. Application to VHDL

VHDL [20] is so far the only standardised hardware description language, and is of increasing popularity in the world of chip design.
Designed to cover various levels and types of description, it is a very complex language that needs to be restricted to be verifiable by Model Checkers [15,16].
A short presentation of the verifiable subset is given, followed by an application example.

5.1 Presentation of the synchronous subset

VHDL is asynchronous, without a predefined master clock, and has neither predefined signal types for memory elements, nor special constructs to describe automata that are often used to specify finite state machines.
For all these reasons, it is not possible to provide a pure syntactical subset of VHDL that describes synchronous sequential circuits. Therefore, a description style is provided for this kind of circuit [22] .
Using this style, the circuits may easily be given semantics in terms of synchronous machines [31] and translated into a symbolic model of a finite state machine [16].

Basically, a design entity should have an input, say `CLK`, of some predefined type `CLOCK`, and the state variables are target signals of either:

- one process, whose first instruction is a wait statement on the rising edge of the clock (e.g. `wait on CLK until (CLK = '1')`),
- some concurrent guarded signal assignments, where the guard expression is
  `CLK = '1' and not CLK'STABLE`.

State variables are modelled by means of signals of type `BIT`, `BIT_VECTOR` or any enumerated type.
The use of variables in processes and procedures is restricted by conditions which guarantee that they have no memory effects.
The use of signal attributes other than 'stable on the master clock' is forbidden.
For a complete description of the syntactic and semantic guidelines, please refer to [22].

5.2 How are TL properties embedded in VHDL descriptions?

We state that TL properties must be written in the VHDL description, for two major reasons: first, a TL property provides information to the designer, for it is a partial specification of what should or should not happen in the circuit; second, when the VHDL description is translated into the verification format, the TL properties are directly provided to the model checker: it is of an easier use for the designer.

But, on the other hand, embedding TL properties directly into the VHDL description may modify the simulation semantics of the language, which must not happen.
Therefore, the assert construct with a severity level set to note is used to express the temporal properties to verify[3].
TL modalities have been defined as functions in a special purpose package `TL_PROOF`.

5.3 Example

Our example is based on a controller of the some shared resource between two clients, presented in the landmark paper of Clarke *et al.* [11]. Both clients `i` may be in three different modes: the Non-critical mode `Ni`, the Critical mode `Ci`, the Trying mode `Ti`.

A possible VHDL description of this automaton is shown below. The inputs are the requests and the release of the shared resource, the synchronizing clock signal. The outputs are the grants to the clients. An internal signal `P` represents the mode in which the clients are.

```
use work.TL_PROOF.all;
use work.BOOLEAN_EXT.all; -- Extensions to boolean logic.

entity MUTEX2_CONTROLLER is
   port (REQ: in BIT_VECTOR(1 to 2); -- requests from clients
         CLK: in CLOCK; -- the master clock
         REL: in BIT; -- release from clients
         GR: out BIT_VECTOR(1 to 2)); -- grants to clients
end MUTEX2_CONTROLLER;

architecture ARCH1 of MUTEX2_CONTROLLER is
   type MODE is (NON_CRITICAL, TRYING, CRITICAL);
   type STATE is array (1 to 2) of MODE;
   signal P : STATE;
   attribute ASSERT_INDICATOR of all: label is TL_PROPERTY;
      -- all assertions are temporal logic properties to be checked
      -- ASSERT_INDICATOR is defined in package TL_PROOF.
begin
   process
   begin
      -- Synchronization on the master clock:
```

[3]Another possibility is to put the TL properties in comments, but we believe it is not a good solution, for most commercial analysers are not designed to treat comments.

```
        wait on CLK until (CLK = '1');
        -- Getting and treating the information:
        if (P(1) = NON_CRITICAL) and (REQ(1) = '1') then
           P(1) <=TRYING; end if;
        if (P(2) = NON_CRITICAL) and (REQ(2) = '1') then
           P(2) <=TRYING; end if;
        if (P(1) = CRITICAL) and (REL = '1') then
           P(1) <=NON_CRITICAL;
           if (P(2) = TRYING) then P(2) <=CRITICAL; end if;
        end if;
        if (P(2) = CRITICAL) and (REL = '1') then
           P(2) <=NON_CRITICAL;
           if (P(1) = TRYING) then P(1) <=CRITICAL; end if;
        end if;
        if P = (TRYING,NON_CRITICAL) then P(1) <=CRITICAL; end if;
        if P = (NON_CRITICAL,TRYING) then P(2) <=CRITICAL; end if
        -- We state that P(1) has a higher priority than P(2).
        if P = (TRYING,TRYING) then P(1) <=CRITICAL; end if
    end process;
    -- Granting the resource:
    GR(1) <='1' when (P(1) = CRITICAL) else '0';
    GR(2) <='1' when (P(2) = CRITICAL) else '0';
    --  Properties:
    M: assert (P(1)/=CRITICAL and P(2)/=CRITICAL) or
              (P(1)=CRITICAL and P(2)/=CRITICAL) or
              (P(1)/=CRITICAL and P(2)=CRITICAL) severity note;
    N_S: assert implies(P(2)=TRYING, AF(P(2)=CRITICAL)) severity note;
    C: assert implies(P(2)=TRYING, AB(P(2)=CRITICAL)) severity note;
    G: assert implies(P(2)=TRYING, EF(P(2)=CRITICAL)) severity note;
end ARCH1;
```

The architecture holds five TL properties, recognizable through the value of a predefined attribute `ASSERT_INDICATOR` for the labels of their *assert* statement.
The *mutual exclusion* property `M` is true whenever the resource is granted to at most one client. The *no starvation* property `N_S` is valid if, when the client 2 requests the resource, it will surely be granted in the future. The property `C` is valid if, when the client 2 is being granted the resource, it has surely been requested before; it expresses the *correctness* of the grant. This is a class of property that can be expressed by means of past-oriented temporal operators. The property `G` is valid, if when the client 2 requests the resource, there exists a computation sequence that leads to a grant.

5.4 Implementation and results

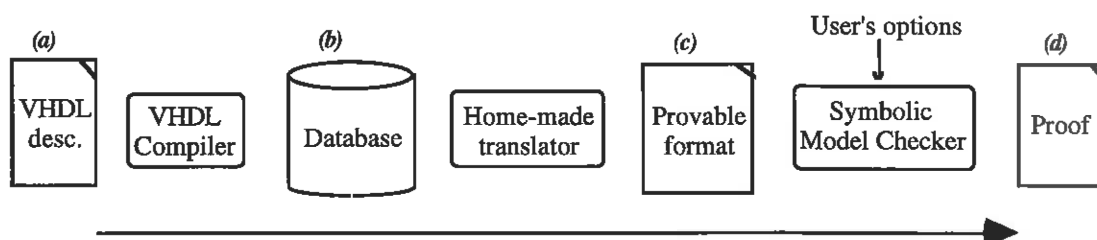

Figure 2: The different pieces of the software

A prototype software tool for the model checking of synchronised sequential circuits described in a VHDL synchronous subset has been implemented. It is decomposed in several compounds (see figure 2):

- A commercial VHDL compiler[4] that takes as input a VHDL description (a) and produces an enriched syntax tree in a database (b) .
- A home-made translator written in C that, via requests to the database (b) , checks the synchronous requirements for the description, and produces a text file (c) that holds, in a provable format (i.e. LISP-like lists of properties), the description of the circuit.
- The symbolic model checker is written in Common LISP, and uses a C-version of the BDD package [19]. It takes as input the provable-format file (c) , and according to the choices of the user, does a custom proof (d) . The options let the user choose the form of the result (a truth table or a Boolean formula), the semantics of the formula (initial or global), and the print out of the reachable states.

Performance results of this system on the example of section 3 are now discussed. All experiments have been performed on a Sun SparcStation 10 in a shared environment. Table 1 holds the data collected on a series of descriptions of mutex controllers with an increasing number of clients.
The first column shows the number of clients that compete for the resource, the second column shows the time needed to compute the set of reachable states. The last four columns show the time needed to compute the set of states where the different temporal properties hold (times are in seconds).

Clients	Reachable states	M	N_S	C	G
2	0.033	0.000	0.017	0.033	0.067
3	0.117	0.083	0.083	0.150	0.100
4	0.467	0.067	0.433	0.133	0.467
5	1.200	0.183	1.250	0.600	1.300
6	3.683	0.783	4.467	2.233	4.617
7	13.083	1.167	10.783	9.050	11.400

Table 1: Experimental results of the Symbolic Model Checker

These results show that computation involving past and future operators have the same complexity. This complexity strongly depends on the size of the circuit and on the number of fix point operators (*EU*, *ES*, and so forth) present in the property.

6. Conclusion

In this paper, we have presented the essential features of symbolic model checking technology, and its implementation using a BDD package for the manipulation of boolean expressions. We have shown that, in a branching time temporal logic, the availability of past-oriented modalities in addition to the usual future-oriented ones gives more expressive power to specify the expected circuit properties. This added versality does not induce any penalty in model checking time.

[4] The analyser called VHDL Tool Integration Platform (VTIP©).

We have defined a VHDL subset and writing style which guarantees that a circuit description can be modelled as a finite state machine. Properties can be asserted about the circuit, using the operators and modalities of the temporal logic, defined in a special purpose package. This makes symbolic model checking available as a formal validation tool on initial specifications of the circuit behavior. We expect this technique to join simulation and tautology checking in CAD verification environments.

Two problems remain to be solved before model checking is widely accepted among designers. The first problem relates to readability of TL. Complex TL formulas involving nested invocations of temporal modalities are difficult to understand. To circumvent this lack of user-friendliness, the use of timing diagrams as a user interface to express properties has been advocated [32], but here again non-trivial properties require the drawing of complex diagrams. A systematic study of the categories of properties typically found in circuit design should head toward providing 'property schemata' that could be taken off the shelf by designers, and automatically compiled in TL formulas.

The other problem is that the fix point computation algorithms are exponential. This limits the applicability of the method to control-oriented circuits with a small number of registers. Property verification for large data paths (as can be found in current processors) is out of reach with symbolic model checking at bit or bit-vector level. Several teams are investigating the possible co-operation of theorem proving and model checking; to our knowledge, automatic mechanized combinations of the two techniques have not yet been produced.

Acknowledgements
This work has been partially supported by the CEC ESPRIT CHARME project number 3216 and the CHARME2 Working Group number 6018.
The authors are grateful to their partners of CHARME for many enlightening discussions and fruitful co-operation over several years. They are particularly thankful to Hans Eveking and Stefan Höreth who gave us their BDD package and provided useful advice and feedback.

References

[1] R.Airiau, J.-M.Bergé, V. Olive and J. Rouillard. *VHDL du langage à la Modélisation*. Collection informatique. Presses polytechniques et universitaires romandes, first edition, 1990. ISBN 2-88074-191-2 (in French).

[2] S. Akers. *Binary Decision Diagram.* IEEE Transactions Computers, C(27), pages 509-516, June 1978.

[3] B. Berkane. *Vérification des systèmes matériels numériques séquentiels synchrones : Application du langage Lustre et de l'outil de vérification Lesar.* PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, Oct. 1992 (in French).

[4] S. Bose and A.L. Fisher. *Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic.* In [21] pages 151-158.

[5] R. Bryant. *Graph-based algorithms for Boolean function manipulation.* IEEE Transactions Computers, C(35), pages 1035-1044, 1986.

[6] R. Bryant. *Symbolic Boolean manipulation with ordered binary-decision diagrams.* IEEE Transactions Computers, 24(3), pages 293-318, Sep. 1992.

[7] J.R. Burch, E.M. Clarke, and D.E.Long. *Representing circuits more efficiently in symbolic model checking.* Internal report, Carnegie Mellon University, Pittsburgh, Pa., Nov. 1990.

[8] J.R. Burch, E.M. Clarke, K.L. MacMillan, D.L. Dill, and J. Hwang. *Symbolic Model Checking : 10^{20} states and beyond.* In LICS'90: 5th annual IEEE symposium on Logic In Computer Science, pages 428-439, Philadelphia, Pa., June 1990. IEEE.

[9] K. Butler, D. Ross, R. Kapur, and M. Mercer. *Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams.* Proc. 28th ACM/IEEE Design Automation Conference, pages 417-420, San Francisco, Ca., June 1991.

[10] G. Cabodi and P. Camurati. *Advancements in symbolic traversal techniques.* In G.J. Milne and L. Pierre editors, Correct Hardware Design and Verification Methods, volume 683 of Lecture Notes in Computer Science, pages 155-166, Arles, May 1993. Springer-Verlag.

[11] E.M. Clarke E.M., E.A. Emerson, and A.P. Sistla. *Automatic verification of finite-state concurrent systems using temporal logic specifications.* ACM Transactions On Programming Languages and Systems, 8(2), pages 244-263, Apr. 1992.

[12] O. Coudert, C. Berthet, and J.-C. Madre. *Verification of sequential machines using functional vectors.* In [21], pages 179-196.

[13] O. Coudert, C. Berthet, and J.-C. Madre. *Verification of synchronous sequential machines based on symbolic execution.* In J.Sifakis, editor, Workshop on automatic verification methods for finite state systems, number 407 in Lecture Notes in Computer Science, pages 365-373, Grenoble, France , June 1989. Springer-Verlag.

[14] O. Coudert and J.-C.Madre. *Symbolic computation of the valid states of a sequential machine: algorithms and discussion.* In International workshop on formal methods for correct VLSI design, Miami, Fl., Jan. 1991. ACM/IFIP WG10.2.

[15] A. Debreil and D. Jaillet. *Synchronous description in VHDL for formal proof and resulting guidelines proposed by BULL.* Advanced Report BULL/92.0001 rev.A. BULL Produits Systèmes Département Développement Assisté, Jul. 92.

[16] D.Déharbe and D. Borrione. *Symbolic model checking of VHDL design entities.* Research Report 925-I, Lab. IMAG-ARTEMIS, Université J. Fourier, Grenoble, France, Dec. 1993.

[17] E. Emerson. *Handbook of theoretical computer science*, volume 2, chapter *Temporal and modal logic*, pages 996-1071. Elsevier Science Publishers B.V., Larsen K.G. and A. Skou editors, 1990.

[18] S. Graf. *Logique du temps arborescent pour la spécification et la preuve de programmes*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, Feb. 1984 (in French).

[19] S. Höreth. *Improving the performances of a BDD-based tautology-checker*. In P.Prinetto and P.Camurati, editors, Correct Hardware Design Methodologies, pages 377-384, Torino, Italy, June 1991. ESPRIT BRA CHARME, North-Holland.

[20] IEEE. *IEEE Standard VHDL Language Reference Manual*, Std 1076-1993, 1993.

[21] IFIP WG 10.2/WG 10.5. volume *VLSI Design Methods-II*, Houthalen, Belgium, 1989. North-Holland, 1990.

[22] C. Le Faou and L. Pierre and A. Salem. *A user-oriented presentation of PREVAIL: a proof environment for VHDL descriptions*. Technical Report RT-71. IMAG, Grenoble, France, Sep. 1991.

[23] P. Lescanne. *Computer experiments with the REVE term rewriting system generator*. In10th ACM symposium on principles of programming languages, pages 99-108, Austin, Tx., 1983.

[24] O. Lichtenstein, A. Pnueli, and L. Zuck. *The glory of the past*. Volume 193 of Lecture Notes in Computer Science, pages 196-218. Springer-Verlag, Brooklyn, Ny., R. Parikh edition, June 1985.

[25] J.-C. Madre. *PRIAM: Un outil de vérification formelle des circuits intégrés digitaux*. PhD thesis, Ecole nationale supérieure des télécommunications, Paris, France, June 1990 (in French).

[26] Z. Manna and A.Pnueli. *A hierarchy of temporal properties*. Research Report STAN-CS-87-1186, Stanford University, Stanford, Ca., Oct. 1987.

[27] M. Newman. *On theories with a combinatorial definition of 'equivalence'*. Annals of mathematics, 43(2), pages 223--243, 1942.

[28] A. Pnueli. *The temporal logic of programs*. In 18th annual IEEE Symposium on Foundations in Computer Science, pages 46-57, 1977.

[29] J.-P. Queille and J.Sifakis. *Specification and verification of concurrent systems in CESAR*. In 5th international symposium on programming, volume 137 of Lecture Notes in Computer Science, pages 244-263, New York, 1981. Springer Verlag.

[30] F. Rocheteau. *Extension du langage LUSTRE à la conception de circuits:le langage LUSTRE-V4 et le système POLLUX*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, Oct. 1992 (in French).

[31] A. Salem. *Vérification formelle des circuits digitaux décrits en VHDL*. PhD Thesis, IMAG, Grenoble, France, Oct. 1992 (in French).

[32] R. Schlör and W. Damm. *Specification and verification of system level hardware designs using timing diagrams*. Proc EDAC/EUROASIC 93. Paris, France, 22-25 Feb. 1993, IEEE Computer Society Press, pages 518-524.

[33] G. Thuau and B. Berkane. *A unified framework for describing and verifying hardware synchronous sequential circuits*. Formal Methods in System Design, 2(3), 1993. ISSN: 0925-9856.

[34] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli. *Implicit state enumeration using BDD's*. Research Report, University of California, Berkeley, Ca., 1990.