Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams

Olivier Coudert

Jean Christophe Madre

Christian Berthet

Bull Research Center
P.C. 58B1
68, Route de Versailles
78430 Louveciennes, FRANCE

Fax: +33 1 39 02 49 77

Tel: +33 1 39 02 42 11 ext. 6528

E-mail: coudert@crg.bull.fr

Telex: 698405F

Submitted to *Workshop on Computer-Aided Verification,*

Rutgers, June 1990.

Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams

Olivier Coudert
Jean Christophe Madre
Christian Berthet

Bull Research Center
P.C. 58B1
68, Route de Versailles
78430 Louveciennes, FRANCE

Abstract

This paper shows that the algorithms on boolean functions that have been developed for proving the equivalence of finite state machines can be used to verify properties expressed in temporal logic. The paper gives a verification procedure of such formulas that does not require the building of the state-transition graph of the machine. This procedure is based on the usual boolean operations in addition with the resolution of a special boolean equation common to all the kinds of formulas. The paper gives a technique for reducing the cost of the resolution procedure of this special equation, which is the bottleneck in the verification.

1 Introduction

Within the community of people that work on the verification of sequential machines, the word "verification" has received two quite different meanings. For some people, to verify a machine is to prove that it holds some properties, such as liveness or safety properties [5]. For the others, to verify a machine is to prove that it is correct with respect to its behavioral specification, which is a program written in some high level hardware description language. This problem comes down to proving the behavioral equivalence between two machines [11].

Several techniques have been independently proposed to deal with these two verification problems. Model Checking [5] [12] consists of a unified set of algorithms for the automatic verification of properties expressed in various temporal logics. This verification method requires the partial or total building of the state-transition graph of the machine under verification. This operation, which can be very memory consuming, strongly limits the size of the machines that can be handled.

The situation is slightly different for the other verification problem. It was first proposed to prove the behavioral equivalence between two machines $\mathcal{M}_1$ and $\mathcal{M}_2$ by building the state-transition graph of the product machine $\mathcal{M}_1 \times \mathcal{M}_2$ [11]. Though methods have been proposed to reduce the memory size needed to store this graph, for instance by representing transitions with *binary decision diagrams* (BDD) [13], these methods have the same limitations than those cited in the previous paragraph. More recently Devadas showed [9] that it is not necessary to build this state-transition

graph in order to check that the machines $\mathcal{M}_1$ and $\mathcal{M}_2$ always produce the same outputs. However this technique is based on a double enumeration of the states and the input patterns of the machines, so that in many cases the time needed to traverse the state-transition graph of the machine $\mathcal{M}_1 \times \mathcal{M}_2$ grows exponentially with the number of inputs of $\mathcal{M}_1$ and $\mathcal{M}_2$.

We have presented in [7] a procedure for proving the equivalence of two incompletely specified Mealy machines that is based on symbolic manipulations of boolean functions. The main difference between this procedure and the one given in [9] is that it manipulates set of states instead of individual states, represented either by their characteristic functions of by vectors of boolean functions [7]. The algorithm performs a *breadth-first* instead of a *depth-first* traversal of the state-transition graph.

This paper shows that the concepts and the algorithms that support this proof procedure also apply to the verification of properties expressed in temporal logic. In particular it shows that it is not necessary to build the state graph of the machine $\mathcal{M}$ to check whether $\mathcal{M}$ holds such properties. Part 2 defines the syntax and the semantics of the temporal formulas to verify, and the kind of machines that are handled. Part 3 gives the verification algorithm used in the system and shows that this algorithm rely on the resolution of one boolean equation. Part 4 is dedicated to this resolution, which is the bottleneck of the verification procedure. A technique is proposed to reduce the cost of this resolution, that is based on two boolean operators called "expand" and "restrict". Part 5 gives some experimental results and discusses them.

2 Verification of CTL Formulas

This section describes the inputs of the verification system presented in this paper. It first defines the model of sequential machines handled. It then gives the syntax and the semantics of the temporal formulas to be verified.

2.1 Deterministic Moore Machines

The sequential machine $\mathcal{M}$ to verify is defined by a 7-tuple (R, I, O, λ, δ, Init, Cns) where:

- R is the finite set of boolean registers noted $r_1, r_2, ..., r_n$ of the machine $\mathcal{M}$. A state of the machine is defined by the values of these registers.
- I is the finite set of boolean inputs of the machine.
- O is the finite set of boolean outputs of the machine.
- λ is the output function of the machine. λ actually is a vector of functions, one for each output. For the sake of simplicity, we suppose here that the machine is a deterministic Moore machine so these functions are boolean functions from the set $\{0, 1\}^n$ into $\{0, 1\}$, i. e. the values of the outputs only depend on the values of the registers.

- δ is the transition function of the machine. δ is a vector of n functions, one for each register. These functions are boolean functions from the set $\{0, 1\}^n \times \{0, 1\}^p$ into $\{0, 1\}$, where p is the number of inputs of the machine.
- Init is the set of initial states of the machine. Each of these initial states is defined by the values of the boolean registers of the machine.
- Cns is a set of constraints on the inputs and the registers of the machine.

In the description of the machine, the output function λ and the transition function δ are written in the textual form of the typed decision graph (TDG) representation. The TDG representation is a compact canonical form of boolean functions introduced by Billon in [1]. It is a refinement of the BDD representation. It holds all the remarkable properties of BDD's in addition with the *null-cost negation.* The output and transition functions of the machine are automatically computed by a tool called PRIAM [2] from the behavioral description of the machine written in the hardware description language LDS, which is the language used at Bull for describing the structure and the behavior of synchronous circuits.

The set Init of initial states of the machine is represented by its characteristic function χ_{Init} written in the same form. By definition of the characteristic function, any state $s = [y_1\ y_2 \dots y_n]$ is in Init iff $\chi_{Init}(s) = \chi_{Init}(y_1, y_2, \dots, y_n) = 1$. The set Init is specified by the designer of the machine. The set Cns is made of constraints given by the designer and of additional constraints computed by PRIAM. This set is also represented by its characteristic function χ_{Cns}: the input $p = [x_1\ x_2 \dots x_p]$ is accepted by the machine in the state $s = [y_1\ y_2 \dots y_n]$ iff $\chi_{Cns}(s, p) = \chi_{Cns}(y_1, y_2, \dots, y_n, x_1, x_2, \dots, x_p) = 1$. Cns can be used to describe an *uncompletely specified* machine, by forbidding the combinations of states and inputs for which the outputs or the next state of the machine are not defined.

2.2 State Formulas in Computation Tree Logic

The temporal formulas handled by the verification system are the state formulas of the computation tree logic CTL [5]. This logic is a formalism proposed by Clarke and Emerson for expressing properties about the states and the computation paths of finite state systems. The meaning of a state formula is relative to a state of the machine, which is defined by the values of its registers. The different state formulas of CTL and their meaning with respect to a state of the machine (R, I, O, λ, δ, Init, Cns) are the following:

1. $r_1, r_2, \dots, r_n$ are state formulas. For any state s of the machine, $s \models r_i$ if and only if the value of the register r_i is 1 in the state s.
2. If f and g are state formulas then so are $(\neg f)$, $(f \wedge g)$, $(f \vee g)$, $(f \Leftrightarrow g)$, and $(f \Rightarrow g)$. The logical connectors have their usual meaning, for instance $s \models (f \wedge g)$ iff $s \models f$ and $s \models g$.
3. If f is a state formula, so is the "all next times" formula $AX(f)$. For any state s, $s \models AX(f)$ iff for any input pattern p that the machine accepts in the state s, then $\delta(s, p) \models f$.

4. If f is a state formula, so is the "some next time" formula $EX(f)$. For any state s, $s \models EX(f)$ iff there exists one pattern p accepted by the machine in the state s, such that $\delta(s, p) \models f$.
5. If f and g are state formulas, so is the formula $A[f \, U \, g]$. For any state s of $\mathcal{M}$, $s \models A[f \, U \, g]$ iff for all paths $(s_0, s_1, ...)$ such that $s_0 = s$, $\exists i((s_i \models g) \wedge (\forall j(0 \leq j \leq i \Rightarrow s_j \models f)))$.
6. If f and g are state formulas, so is the formula $E[f \, U \, g]$. For any state s, $s \models E[f \, U \, g]$ iff there exists a path $(s_0, s_1, ...)$, $s_0 = s$, and $\exists i((s_i \models g) \wedge (\forall j(0 \leq j \leq i \Rightarrow s_j \models f)))$.

For the sake of clarity, some connectors have been added that can be expressed in term of the preceding ones. For any state formula f, $AF(f) =_{def} A[True \, U \, f]$, i. e. on every path there exists a state on which f holds; $EF(f) =_{def} E[True \, U \, f]$, i. e. there exists a state on some path on which f holds; $AG(f) =_{def} \neg (EF(\neg f))$, i. e. f holds on any state of every path; $EG(f) =_{def} \neg (AF(\neg f))$, i. e. there exists a path on which f holds at any state.

3 Verification Algorithm

There is a class of state formulas which can be checked with the procedure that proves the equivalence of two machines [8]. The machine $\mathcal{M}$ holds a state formula f belonging to this class iff $\mathcal{M}$ is equivalent to some finite state automaton $\mathcal{A}_f$ directly obtained from the formula f. In this case the proof is made in a forward chaining way. This part presents a more general proof algorithm that takes into account all the state formulas described in the previous section. This algorithm, that works in a backward chaining way, does not require the building of the state-transition graph of the machine $\mathcal{M}$ under verification.

The algorithm takes as inputs a machine $\mathcal{M}$ described by its 7-tuple (R, I, O, λ, δ, Init, Cns) and the formula f to be verified. It recursively computes the set of states of the machine $\mathcal{M}$ that satisfy the formula f from the sets of states that satisfy its subformulas. At each step there are only six basic cases to consider that correspond to the 6 kinds of formulas given in Section 2.2. Once the set of states F that satisfy the whole formula is obtained, to check whether $s \models f$ for some state s comes down to checking whether s is in F.

Sets of states are represented by the TDG's of their characteristic functions. This makes the formulas of type (1) and (2) trivial to treat. For instance, if $f = (f_1 \wedge f_2)$, and χ_{F1} and χ_{F2} are the characteristic functions of the sets of states that satisfy f_1 and f_2 respectively, then χ_F is $\chi_{F1} \wedge \chi_{F2}$. The cost of computing the TDG of χ_F is in $O(|G_1| \times |G_2|)$ where G_1 and G_2 are the TDG's of χ_{F1} and χ_{F2} respectively, and $|G|$ is the number of vertices in the TDG G. In the same way, for any state s, $s \models f$ iff $\chi_F(s) = 1$, which is the case iff the path defined by s in the TDG of χ_F leads to a leaf equal to True. Finally, M, Init $\models f$ iff $(\chi_{Init} \Rightarrow \chi_F)$ is a tautology. The other basic cases are more difficult to handle and require specific algorithms based on boolean equation solving [7].

3.1 AX and EX formulas

Let f be a formula and F be the set of states that satisfy f. The set of states AX that satisfy $AX(f)$ is defined by the equation $AX = \{s \,/\, \forall p, \delta(s, p) \in F\}$ and the set EX of states that satisfy $EX(f)$ is defined by the equation $EX = \{s \,/\, \exists p, \delta(s, p) \in F\}$. The characteristic functions of these sets are the following:

$$\chi_{AX}(s) = (\forall p\ \chi_{Cns}(s, p) \Rightarrow \chi_F(\delta(s, p))), \text{ and} \tag{1}$$

$$\chi_{EX}(s) = (\exists p\ \chi_{Cns}(s, p) \wedge \chi_F(\delta(s, p))). \tag{2}$$

3.2 AU and EU formulas

Let f and g be two formulas and F and G be the sets of states that satisfy f and g respectively. The set AU of states of the machine $\mathcal{M}$ that satisfy the formula $A[f\ U\ g]$ is defined as the limit of the following converging sequence of sets [3]:

- $A_0 = G$, and
- $A_{k+1} = A_k \cup \{s \,/\, (\chi_F(s) = 1) \wedge \forall p\ (\chi_{Cns}(s, p) = 1) \Rightarrow (\chi_{Ak}(\delta(s, p)) = 1)\}$.

The characteristic functions of the different sets of this sequence are the following:

- $\chi_{A0}(s) = \chi_G(s)$, and
- $\chi_{Ak+1}(s) = \chi_{Ak}(s) \vee (\chi_F(s) \wedge (\forall p\ \chi_{Cns}(s, p) \Rightarrow \chi_{Ak}(\delta(s, p))))$. (3)

In the same way the set EU of states of the machine $\mathcal{M}$ that satisfy the formula $E[f\ U\ g]$ is defined as the limit of the following converging sequence of sets:

- $E_0 = G$, and
- $E_{k+1} = E_k \cup \{s \,/\, (\chi_F(s) = 1) \wedge \exists p\ ((\chi_F(s) = 1) \wedge (\chi_{Cns}(s, p) = 1)) \wedge (\chi_{Ek}(\delta(s, p)) = 1)\}$.

The characteristic functions of the different sets of this sequence are the following:

- $\chi_{E0}(s) = \chi_G(s)$, and
- $\chi_{Ek+1}(s) = \chi_{Ek}(s) \vee (\chi_F(s) \wedge (\exists p\ \chi_{Cns}(s, p) \Rightarrow \chi_{Ek}(\delta(s, p))))$. (4)

3.3 The Critical Boolean Equation

In the formulas (1) to (4) given in the preceding sections appear two predicates that actually are the same one, and that has the form $(\exists p\ \chi_1(s, p) \wedge \chi_2(\delta(s, p)))$. Indeed, since the formula $(\forall x\, f)$ is equivalent to $(\neg (\exists x\, \neg f))$, the equation (1) can be rewritten into

$$\chi_{AX}(s) = \neg\ (\exists p\ \chi_{Cns}(s, p) \wedge \neg\ \chi_F(\delta(s, p))), \tag{1'}$$

and the equation (3) into

$$\chi_{Ak+1}(s) = \chi_{Ak}(s) \vee (\chi_F(s) \wedge \neg\ (\exists p\ \chi_{Cns}(s, p) \wedge \neg\ \chi_{Ak}(\delta(s, p))). \tag{3'}$$

From the computational point of view this means that the 6 basic cases we have to deal with can be treated with the standard boolean operations (negation, conjunction, and disjunction) in addition with a resolution procedure of the equation $(\exists p\ \chi_1(s, p) \wedge \chi_2(\delta(s, p)))$. The standard boolean operations are efficiently supported by the TDG representation [2]. The TDG representation is canonical so it is trivial to detect that the sequences (A_i) and (E_i) converge, since $\chi_{Ak+1} = \chi_{Ak}$ iff their TDG's are isomorph.

The resolution procedure of the equation $(\exists p\ \chi_1(s, p) \wedge \chi_2(\delta(s, p)))$ is the bottleneck of the proof procedure. This resolution procedure works in two steps. The first step consists in building the TDG G_e of the formula $(\chi_1(s, p) \wedge \chi_2(\delta(s, p)))$. There is a simple way to obtain G_e that consists in building the TDG G_r of the transition relation of the machine. N dummy variables $y_1', y_2', \ldots y_n'$ are introduced, and G_r is the TDG of the formula $\wedge_j (y_j' \Leftrightarrow \delta_j(s, p))$. The TDG G_e can be obtained from the TDG G_r by cutting in this TDG all the branches that represent states that are not in the set denoted by χ_2. The problem, that we already mentioned in [6], is that for complex machines it is not possible to build the TDG G_r. Part 4 proposes a resolution procedure that does not require the building of this TDG.

The second step in the resolution is the elimination of the existentially quantified input variables from the TDG G_e. This elimination uses the following property that is an immediate consequence of the definition of a boolean formula:

$(\exists x_1\ f(x_1, x_2, \ldots, x_n)) \Leftrightarrow (f(0, x_2, \ldots, x_n) \vee f(1, x_2, \ldots, x_n))$.

Since the elimination of the input variables from the graph G_e reduces the number of its vertices, nothing can prevent this elimination to terminate.

4 Computing the Critical Equation

The first step of the resolution procedure consists in building the TDG G_e of the formula $(\chi_1(s, p) \wedge \chi_2(\delta(s, p)))$. This part describes a technique that can be used to compute this TDG when the TDG of the transition relation of the machine is too large to be built.

The term $\chi_2(\delta(s, p))$ can be computed using the "composition" algorithm proposed by Bryant in [4]. The cost of this operation is related to the number of vertices in the TDG's of the transition functions $\delta_1, \delta_2, \ldots, \delta_n$. Experience shows that for complex machines for which these functions have several thousands of vertices it is not possible to compute the term $\chi_2(\delta(s, p))$ with this standard method because the resulting TDG becomes too large.

The technique proposed here to compute the TDG G_e is based on a new boolean operator called "restrict" and an original composition function called "expand". The "restrict" operator allows us to reduce the number of vertices in the TDG's of the functions used in the term $\chi_2(\delta(s, p))$. The "expand" function allows us to replace the building of the TDG that represents $\chi_2(\delta(s, p))$ with the building of a set of smaller TDG's from which the input variables can be independently eliminated.

4.1 The restrict Operator

This section presents a boolean operator called "restrict" that gives a means to reduce the size of the TDG's of the functional vector $\delta(s, p)$ used in the formula $\chi_1(s, p) \wedge \chi_2(\delta(s, p))$. This directly reduces the cost of computing the TDG of the formula $\chi_2(\delta(s, p))$.

The idea that led to the "restrict" operator is made clear by the following remark. The transition function δ of the machine is a vector of boolean functions. This function gives for any state and any input the state that is attained at the next cycle of the machine. Yet the set we are interested in, denoted by the formula $(\exists p\ \chi_1(s, p) \wedge \chi_2(\delta(s, p)))$, is the set of states s *that satisfy the constraint represented by* χ_1 and have at least one successor in the set represented by χ_2. This means that in the formula $(\chi_1(s, p) \wedge \chi_2(\delta(s, p)))$ the transition function δ can be replaced by any other function δ' that is equal to δ on the domain denoted by χ_1. This is exactly what is done by the "restrict" operator which finds such a functional vector δ' that has the following remarkable property: each of its component has a TDG with less vertices than the corresponding component of the functional vector δ.

```
(defstruct vertex root low high mark info)
(defvar True)
(defvar False)

(de restrict (vtx cns)
  (cond
    ((eq cns False)
     (print "ERROR IN RESTRICT : Inconsistency Constraint")
      False)
    (t
     (restrict-aux vtx cns))))

;; vtx is the BDD that must be restricted.
;; cns is the BDD that represent the constraint.
(de restrict-aux (vtx cns)
  (cond
    ((or (eq vtx True) (eq vtx False) (eq cns True)) vtx) ;; (0)
    ((< (order (root vtx)) (order (root cns)))            ;; (1)
     (norme-vtx (root vtx)
                (restrict-aux (low vtx)  cns)
                (restrict-aux (high vtx) cns)))
    ((= (order (root vtx)) (order (root cns)))            ;; (2)
     (cond
       ((eq (low cns)  False)                             ;; (2.1)
        (restrict-aux (high vtx) (high cns)))
       ((eq (high cns) False)                             ;; (2.2)
        (restrict-aux (low vtx)  (low cns)))
       (t                                                 ;; (2.3)
        (norme-vtx (root vtx)
                   (restrict-aux (low vtx)  (low cns))
                   (restrict-aux (high vtx) (high cns))))))
    ((> (order (root vtx)) (order (root cns)))            ;; (3)
     (restrict-aux vtx (disj (low cns) (high cns))))))
```

Figure 1. The Function restrict .

A logical definition of this operation would be too costly for this paper, so we give an algorithmic definition of the *restrict* operator. Figure 1 gives the lisp function `restrict` that takes as inputs the BDD's of a function f and of a constraint c, and returns the BDD of a function f' that is equal to f on the domain defined by c. We consider this algorithm as the definition of the *restrict* operator.

Theorem 1. *Let f and c be two boolean formulas, such that c is satisfiable. Then the formula* $(c \Rightarrow (restrict(f, c) \Leftrightarrow f))$ *is a tautology.*

Proof. The proof is made by induction on the number of variables occurring in the BDD's of D_f and D_c of the formulas f and c respectively. Consider first that there are no variables in D_f and D_c. Then since c is not an antilogy, c is equal to True. In this case (case (0) of Figure 1), the algorithm states that $restrict(f, c) = f$.

The induction hypothesis is that $(c \Rightarrow (restrict(f, c) \Leftrightarrow f))$ is a tautology for any formula f and any satisfiable formula c whose BDD's use at the most n variables $x_1, ..., x_n$. Now consider a formula f and $c \neq$ False such that the BDD's of f and c use the variable x_0 and some of the variables $x_1, ..., x_n$. If D_f is either True or False or if D_c is True then the situation is the same as above, so we will suppose that both D_f and D_c are not trivial. Without loss of generality we will also suppose that the order of the variable x_0 is less than the orders of all the other variables. Then there are three cases to consider:

(1) The variable x_0 occurs in D_f but not in D_c. This means that the variable x_0 appears at the top of D_f but not at the top of D_c. If Shannon's expansion of f with respect to the variable x_0 is noted $(\neg x_0 \wedge f_F) \vee (x_0 \wedge f_T)$ then the algorithm returns the BDD of the function $(\neg x_0 \wedge restrict(f_F, c)) \vee (x_0 \wedge restrict(f_T, c))$. Since the variable x_0 does not occur in the low and high branches of the BDD D_f, the induction hypothesis states that $(c \Rightarrow restrict(f_F, c) \Leftrightarrow f_F)$ and $(c \Rightarrow restrict(f_T, c) \Leftrightarrow f_T)$ are tautologies. The variable x_0 does not occur in D_c, which means that the value of c does not depend on the value of x_0, so it is immediate by substitution that the formula

$$(c \Rightarrow (((\neg x_0 \wedge f_F) \vee (x_0 \wedge f_T)) \Leftrightarrow (\neg x_0 \wedge restrict(f_F, c)) \vee (x_0 \wedge restrict(f_T, c)))$$

is a tautology. This means that the formula $(c \Rightarrow (restrict(f, c) \Leftrightarrow f))$ is a tautology.

(2) The variable x_0 occurs in D_f and in D_c. This means that x_0 is at the top of D_f and D_c. We note $f = (\neg x_0 \wedge f_F) \vee (x_0 \wedge f_T)$ and $c = (\neg x_0 \wedge c_F) \vee (x_0 \wedge c_T)$ Shannon's expansions of f and c with respect to x_0 respectively. There are three subcases to consider:

(2.1) c_F = False. Then $c = (x_0 \wedge c_T)$, and the function returns $restrict(f_T, c_T)$. Since c is satisfiable then c_T is different from False and the function $restrict(f_T, c_T)$ is well defined. The variable x_0 does not occur in the BDD's of c_T and f_T so the induction hypothesis holds: $(c_T \Rightarrow (restrict(f_T, c_T) \Leftrightarrow f_T))$ is valid. This means that the formula $((x_0 \wedge c_T) \Rightarrow (restrict(f_T, c_T) \Leftrightarrow f_T))$ is also valid. By definition of Shannon's expansion, the formula $(x_0 \Rightarrow (f \Leftrightarrow f_T))$ is also valid. Finally, by

substitution in the preceding formula, we obtain that the formula $(c \Rightarrow (restrict(f, c) \Leftrightarrow f))$ is a tautology.

(2.2) c_T = False. Then $c = (x_0 \wedge c_F)$. This case is treated in the same way than the case 2.1 with the difference that the function restrict returns the function $restrict(f_F, c_F)$.

(2.3) $c_F \neq$ False and $c_T \neq$ False. In this case the algorithm states that the resulting function is $restrict(f, c) = (\neg x_0 \wedge restrict(f_F, c_F)) \vee (x_0 \wedge restrict(f_T, c_T))$. The induction hypothesis can be used to say that the formulas $(c_F \Rightarrow (restrict(f_F, c_F) \Leftrightarrow f_F))$ and $(c_T \Rightarrow (restrict(f_T, c_T) \Leftrightarrow f_T))$ are valid. It is immediate by subsitution that $(c_F \Rightarrow ((\neg x_0 \wedge restrict(f_F, c_F)) \Leftrightarrow (\neg x_0 \wedge f_F)))$ is valid, and so is $(c_F \Rightarrow ((\neg x_0 \wedge restrict(f_F, c_F)) \Leftrightarrow (\neg x_0 \wedge f_F)))$. Morever we trivially have the tautology $(\neg x_0 \Rightarrow (\neg (x_0 \wedge restrict(f_T, c_T)) \wedge \neg (x_0 \wedge f_T)))$. So we obtain $((\neg x_0 \wedge c_F) \Rightarrow (((x_0 \wedge restrict(f_T, c_T)) \vee (\neg x_0 \wedge restrict(f_F, c_F))) \Leftrightarrow ((x_0 \wedge f_T) \vee (\neg x_0 \wedge f_F))))$. In the same way, we have the trivial tautology $(x_0 \Rightarrow (\neg (\neg x_0 \wedge restrict(f_F, c_F)) \wedge \neg (\neg x_0 \wedge f_F)))$, and we obtain $((x_0 \wedge c_F) \Rightarrow (((\neg x_0 \wedge restrict(f_F, c_F)) \vee (x_0 \wedge restrict(f_T, c_T))) \Leftrightarrow ((\neg x_0 \wedge f_F) \vee (x_0 \wedge f_T))))$. These two tautologies imply that $(((\neg x_0 \wedge c_F) \vee (x_0 \wedge c_T)) \Rightarrow (((\neg x_0 \wedge restrict(f_F, c_F)) \vee (x_0 \wedge restrict(f_T, c_T))) \Leftrightarrow ((\neg x_0 \wedge f_F) \vee (x_0 \wedge f_T))))$ is valid, that is $(c \Rightarrow (restrict(f, c) \Leftrightarrow f))$ is a tautology.

(3) The variable x_0 occurs in D_c but not in D_f. Shannon's expansion of c with respect to x_0 is $c = (\neg x_0 \wedge c_F) \vee (x_0 \wedge c_T)$ and the algorithm returns the function $restrict(f, c) = restrict(f, c_F \vee c_T)$. The variable x_0 does not occur in the BDD's of $(c_F \vee c_T)$ so the induction hypothesis can be used to state that $((c_F \vee c_T) \Rightarrow (restrict(f, c_F \vee c_T) \Leftrightarrow f))$ is valid. The formulas $((\neg x_0 \wedge c_F) \Rightarrow c_F)$ and $((x_0 \wedge c_T) \Rightarrow c_T)$ are trivially valid. These two tautologies imply that $((\neg x_0 \wedge c_F) \vee (x_0 \wedge c_T)) \Rightarrow (c_F \vee c_T)$ is valid, which gives the result. ❑

Theorem 2. *Let f and c be two boolean formulas with c* $\neq$ False. *The BDD of restrict(f, c) has less vertices than the BDD of f.*

Proof. The proof is made by induction on the size of the BDD of f. The idea used in the proof is that the algorithm recursively makes a copy of the BDD of f, except for the case (2.1) and (2.2) where one vertex in this BDD is eliminated. ❑

Theorem 3. *Let f and c be two boolean formulas with c* $\neq$ False. *If f is equivalent to c, then restrict(f, c)* = True. *If f is equivalent to* $\neg$ *c, then restrict(f, c)* = False.

Proof. If f is equivalent to c, then $f = c$. By induction on the size of the BDD of c it can be shown that the only cases used in the algorithm are cases (0) and (2), and that each recursion returns the constant True. This means that $restrict(c, c)$ = True. This is the same for $f = \neg c$. ❑

There is a similar function *restrict* that works on TDGs. It uses a cache to avoid redundant computations so that its complexity is in $O(|G_f| \times |G_c|)$ where $|G|$ is the number of vertices in the TDG G. It also uses Theorem 3 to avoid useless recursions.

Thanks to Theorem 1, the formula $\chi_1(s, p) \wedge \chi_2(\delta(s, p))$ is equal to $\chi_1(s, p) \wedge \chi_2(restrict(\delta(s, p), \chi_1(s, p)))$. Theorem 2 states that each function $restrict(\delta_k(s, p), \chi_1(s, p))$ has a TDG with less vertices than the TDG of $\delta_k(s, p)$. This means that computing the TDG of the term $\chi_2(restrict(\delta(s, p), \chi_1(s, p)))$ is less costly than computing the TDG of the term $\chi_2(\delta(s, p))$.

4.2 The Function expand

The first step in the resolution of the equation $(\exists p\ \chi_1(s, p) \wedge \chi_2(\delta(s, p)))$ is to compute the TDG G_e that represents the formula $(\chi_1(s, p) \wedge \chi_2(\delta(s, p)))$. The variables that occur in this TDG are the boolean variables associated to the registers and those associated to the input variables of the machine. The second step of the resolution consists in eliminating these input variables in order to obtain the TDG G_f of the characteristic function solution of the equation. The variables that occur in this final TDG are the variables associated to the registers, so this TDG generally has less vertices than the intermediate TDG G_e.

The composition function takes as input the TDG of a function $f(x_1, x_2, ..., x_n)$, the TDGs of n functions $g_1, g_2, ..., g_n$ and it returns the TDG of the function $f(g_1, g_2, ..., g_n)$. For the sake of clarity, a simple implementation of this composition function that works on BDD's is given in Figure 2. It is directly based on the following property which is a direct consequence of Shannon's expansion theorem [4]:

$$f(g_1, g_2, ..., g_n) = ((\neg g_1) \wedge f(0, g_2, ..., g_n)) \vee (g_1 \wedge f(1, g_2, ..., g_n)$$

```
;; computes the composition function f(g1, g2, ..., gn).
;; l is the list ((x1 . g1) ... (xn . gn)).

(de compose (f l)
  (cond
    ((or (eq f True) (eq f False))
     f)
    ((is-marked f)
     (get-info f))
    ((> (order (root f)) (order (caar l)))
     (compose f (cdr l)))
    ((= (order f) (order (car l)))
     (set-mark f)
     (set-info f
               (disj (conj (negate (cdar l))
                           (compose (low f) (cdr l)))
                     (conj (cdar l)
                           (compose (high f) (cdr l))))))))
```

Figure 2. The lisp function compose.

The function `compose` traverses the BDD of the function f in a depth-first manner and computes the resulting BDD in a bottom-up way. It uses a mark to treat each vertex in the BDD of the function f only once. For each vertex there are two cases to consider:

(1) The vertex has already been treated. In this case the function returns the pointer to the resulting BDD stored in the `info` field of the vertex.

(2) The vertex has not already been treated. In this case the function compose is recursively applied on the `low` and `high` branches on the vertex. The resulting BDD is computed using the property given above and stored in the `info` field.

The problem with this composition scheme is that sometimes intermediate BDD's associated to internal vertices of the BDD of the function f become very large and the system runs out of memory, so the result never reaches the top of the BDD.

The idea that underlies the expand function is to express the function $\chi_2(\delta(s, p))$ as a sum of functions whose TDGs have less vertices than the TDG G_e. If these functions are noted $h_1, ..., h_k$, the equation becomes $(\exists p\ \chi_1(s, p) \wedge (\vee_j\ h_j(s, p)))$ that can be rewritten to $(\exists p \vee_j (\chi_1(s, p) \wedge h_j(s, p)))$, by distributing the conjunction over the disjunction. The existential quantifier commutes with the disjunction, so that the equation can be further transformed into $(\vee_j\ (\exists p\ \chi_1(s, p) \wedge h_j(s, p)))$. This final form of the equation expresses that its resolution can be decomposed into a sequence of resolutions of simpler equations.

The computation of a list of formulas $h_1, h_2, ..., h_k$ whose sum is equal to the formula $f(g_1, ..., g_n)$ is made by the function expand. Figure 3 gives the algorithm of this function for BDD's. In addition to the usual fields, each vertex in the BDD D_f of f has two fields named `share` and `path`. The field `share` of a vertex is the number of vertices of D_f that have pointers to this vertex. For any vertex v in the BDD there can exist several paths from the root to this vertex. Each path is uniquely defined by a product, and we note S_v the sum of these products. At the end of the computation, the field `path` of each vertex in the BDD contains the composition $S_v(g_1, ..., g_n)$. This computation is made in a top-down manner that uses the field `share` to freeze the recursive descent until all paths to some vertex have been collected. During the descent, at each time when a leaf True is reached, the BDD of one of the formulas h_j is generated. Since the formula that should be computed is the sum of all the paths to the leaf True, the sum of the formulas h_j is equal to $f(g_1, ..., g_n)$.

```
;; The function expand returns a list of BDD's, whose sum is the
;; BDD of f(g1, ..., gn).  l is the list ((x1 . g1) ... (xn . gn))
;; The functions get-share, get-path, set-share and set-path are
;; used to read or update the fields share and path of a vertex.

(defvar list-disj)

(de expand (f l)
    (update-fields f)  ;; put the fields share and path on each vertices of f.
    (setq list-disj ())
    (expand2 f l True)
    list-disj)

(de expand2 (vtx l path)
  (cond
    ((eq vtx False))
    ((eq vtx True)
     (newl list-disj path))      ;; a new subformula is append to list-disj.
    ((> (order (root vtx)) (order (caar l)))
     (expand2 vtx (cdr l) path))
    ((= (order (root vtx)) (order (caar l)))
     (set-share vtx (decr (get-share vtx)))       ;; update the field share ...
     (set-path  vtx (disj (get-path  vtx) path))  ;; ... and the field path.
     (when (= (get-share vtx) 0)
          ;; test whether all the paths to vtx has been computed
          (expand2 (low vtx)  (cdr l)
                   (conj (negate (cdar l)) (get-path  vtx)))
          (expand2 (high vtx) (cdr l)
                   (conj (cdar l) (get-path  vtx)))))))
```

Figure 3. The lisp function expand.

This process is demonstrated on the BDD drawn in Figure 4. The function compose applied to this BDD would build the BDD of the formula

$$(\neg g_1 \wedge ((\neg g_2 \wedge g_3 \wedge g_4) \vee (g_2 \wedge ((\neg g_3 \wedge g_4) \vee (g_3 \wedge \neg g_4))))) \vee (g_1 \wedge \neg g_2 \wedge \neg g_4)).$$

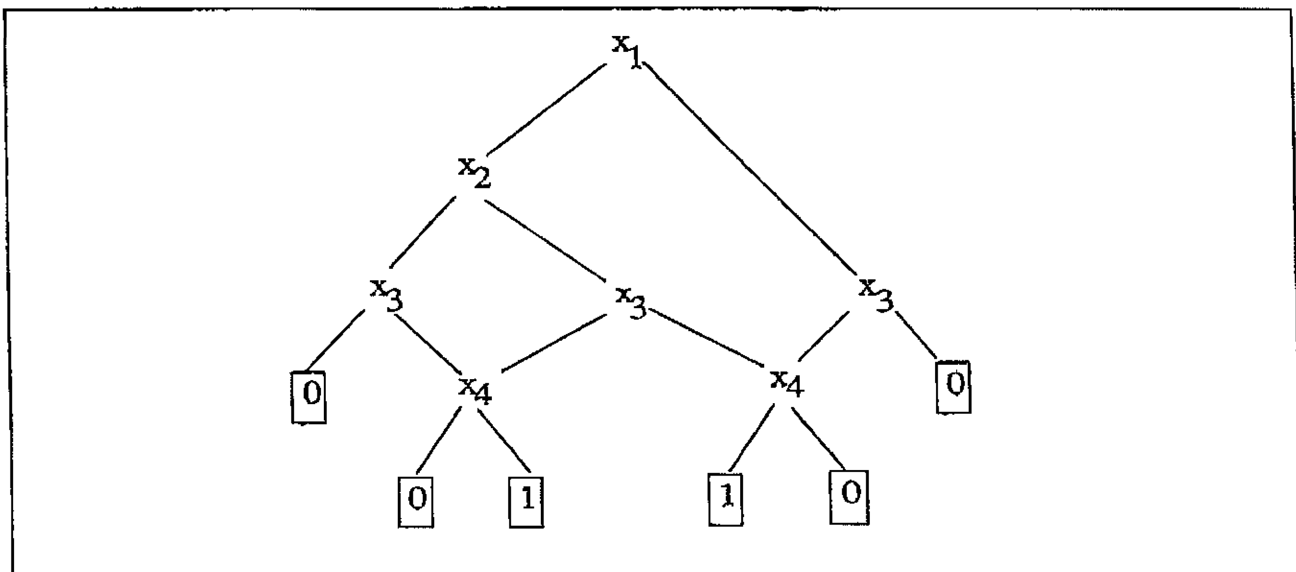

Figure 4. Exemple of a BDD for the Expand Function.

Assuming that each of the BDD's of the formulas g_k have N vertices, the resulting BDD could have N^{12} vertices. The function `expand` generates the partial formulas

$$(g_4 \wedge ((\neg g_1 \wedge \neg g_2 \wedge g_3) \vee (\neg g_1 \wedge g_2 \wedge \neg g_3))), \text{ and}$$

$$(\neg g_4 \wedge ((\neg g_1 \wedge g_2 \wedge g_3) \vee (g_1 \neg g_3))),$$

which cannot have more than N^7 vertices. This means that the time and the memory requirements needed to compute these formulas are much less than those needed to compute the whole formula.

Figure 5 gives the resolution procedure of the critical equation $(\exists p\ \chi_1(s, p) \wedge \chi_2(\delta(s, p)))$. This procedure works in two steps. It first computes the list of formulas $h_1, h_2, ..., h_k$. It then eliminates the inputs variables from each of them and sums the resulting formulas.

```
;; δ is the list (δ1(s, p) ... δrn(s, p)).
;; registers is the list of the rn registers (y1 ... yrn).
;; inputs is the list of the inputs.

(de solve-critical-equation (χ1 χ2 δ registers inputs)
  (let ((res False)
        (partial-formulas
          (expand χ2 (forcar2 δk δ yk registers
                        (cons yk (restrict δk c1))))))
    (foreach f partial-formulas
      (setq res (disj res (exist-abstract (conj χ1 f) inputs))))

    res))
```

Figure 5. Efficient Resolution of the Critical Equation.

5 Experimental Results - Discussion

The verification algorithms given above has been applied to two machines. CLM is a part of an interface board designed at BULL. It is made of 33 state registers, and has 14 inputs. The TDG's that represent its vector of transition functions have more than 10000 vertices and the largest of them has more than 2500 vertices. This machine has more than 377000 valid states out of 2^{33} possible states. The CPU time needed to symbolically traverse the state diagram of this machine using the procedure given in [7] is 2000 seconds on a DPX5000 mini-computer. We did not succeed in computing the TDG of the transition relation of this machine. The property to be proved valid required the computation of only one fixed point that took 38 steps and 4000s of CPU time to be obtained. The term $\chi_2(\delta(s, p))$ of the critical equation stopped to be computable using the standard composition algorithm after a very small number of steps. The restrict function was very useful since it reduced the TDG's used to compute the term $\chi_2(\delta(s, p))$ in such a way that during the backward traversal, none of them had more than 186 vertices. The largest TDG representing one of the sets in the sequence had 352 vertices.

The sequential machine Sync is a synchronizer that has 21 state registers and 4 inputs. The property to be verified was that the value of the output OK_0 is equal to 1 in any valid state of the machine. This is expressed by the formula $s_I \models AG(OK_0 = 1)$, where s_I is the initial state of the machine. The CPU time needed to prove this formula valid was 4160 seconds and the fixed point was found in 9 steps. The restrict operator was not very useful since it did not reduce the sizes of the TDG's of the transition function. The largest TDG found in the sequence of sets had more than 2000 vertices.

The formula $AG(OK_0 = 1)$ belongs to the class of formulas that can be checked in a forward chaining way by traversing the state diagram of the machine using the symbolic traversal procedure given in [6] [7]. At each step during the traversal that starts from the initial state s_I, it is sufficient to check that the output ok_0 is equal to 1. Since the traversal stops when all the valid states of the machine have been reached, the proof is complete. Verifying the property in this manner takes 200 seconds, and the traversal is done in 20 steps. This example seems to show that whenever the formula to be proved belongs to the class of formulas that can be proved using the symbolic traversal procedure, it is better to use this procedure rather than the backward chaining procedure described in this paper.

Conclusion

This paper has presented a procedure that automatically checks whether some deterministic Moore machine $\mathcal{M}$ holds some property expressed in the CTL formalism, without building the state-transition graph of the machine. The verification algorithm procedure is based on the algorithms that were initially developed for proving the equivalence between two machines, in addition with a procedure for solving a special boolean equation common to all the formulas.

Experience shows this resolution procedure is the bottleneck of the verification algorithm. This seems quite understandable if we think about what this resolution procedure does: given a set of states S_F it computes the set of states S_I that have at least one successor in S_F. This is exactly the converse of the operation performed during the symbolic traversal of the machine described in [5], which consists in computing S_F from S_I. That operation already was the bottleneck of that symbolic forward traversal procedure.

The technique proposed here does not give to the symbolic backward traversal procedure the performance of the forward symbolic traversal procedure. This may have several causes. Firstly the symbolic backward traversal procedure manipulates much more states than does the forward symbolic traversal procedure, which only manipulates valid states. This can lead to increase the number of steps needed to find the fixed point for some formula, because of long paths containing invalid states. It could be more interesting to compute once for all the valid states of the machine and then to consider only these states in the backward traversal. Secondly the forward symbolic traversal procedure uses a technique to reduce, at each step, the size of the TDG that represents the

set of states that must be analyzed [6]. This technique cannot be applied here so that the resolution procedure of the backward traversal manipulates larger TDG's than does the one of the forward traversal.

References

[1] J. P. Billon, "Perfect Normal Forms for Discrete Functions", *BULL Research Report N°87019*, 1987.

[2] J. P. Billon, J. C. Madre, "Original Concepts of PRIAM, an Industrial Tool for Efficient Formal Verification of Combinational Circuits", in *The Fusion of Hardware Design and Verification*, G. J. Milne Editor, North Holland, 1988.

[3] S. Bose, A. Fisher, "Automatic Verification of Synchronous Circuits Using Symbolic Logic Simulation and Temporal Logic", in *Proc. of the IFIP International Workshop, Applied Formal Methods for Correct VLSI Design*, Leuven, 1989.

[4] R.E. Bryant, "Graph-based Algorithms for Boolean Functions Manipulation", *IEEE Transactions on Computers*, Vol C35 N°8, 1986.

[5] E. M. Clarke, O. Grumbreg, "Research on Automatic Verification of Finite-State Concurrent Systems", *Annual Revue Computing Science*, vol. 2, pp 269-290, 1987.

[6] O. Coudert, C. Berthet, J. C. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution", in *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, 1989.

[7] O. Coudert, C. Berthet, J. C. Madre, "Verification of Sequential Machines Using Boolean Functional Vectors", in *Proc. of the IFIP International Workshop, Applied Formal Methods for Correct VLSI Design*, Leuven, 1989.

[8] O. Coudert, C. Berthet, J. C. Madre, "Formal Boolean Manipulations for the Verification of Sequential Machines", to appear in *Proc. of the First European Design Automation Conference*, Glasgow, 1990.

[9] S. Devadas, H.K. Ma, and R. Newton, "On the Verification of Sequential Machines at Differing Levels of Abstraction", *IEEE Transactions on CAD*, Vol. 7, No. 6, 1988.

[10] G. J. Holtzman, "Algorithms for Automated Protocol Validation", in *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, 1989.

[11] Z. Kovahi, *Switching and Finite Automata Theory*, McGraw- Hill Book Edition, 1978.

[12] J.P. Queille, J. Sifakis, "Fairness and Related Properties in Transition Systems", *Acta Informatica*, 19, pp 195-220, 1983.

[13] K. J. Supowit, S. J. Friedman, "A new Method for Verifying Sequential Circuits", *Proc. of the 23rd Design Automation Conference*, 1986.