Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams

Olivier Coudert Jean Christophe Madre Christian Berthet

Bull Research Center, PC 62 A13,
68, Route de Versailles
78430 Louveciennes FRANCE

Abstract

This paper presents the algorithm we have developed for proving that a finite state machine holds some properties expressed in temporal logic. This algorithm does not require the building of the state-transition graph nor the transition relation of the machine, so it overcomes the limits of the methods that have been proposed in the past. The verification algorithm presented here is based on Boolean function manipulations, which are represented by typed decision graphs. Thanks to this canonical representation, all the operations used in the algorithm have a polynomial complexity, expect for one called the computation of the "critical term". The paper proposes techniques that reduce the computational cost of this operation.

1 Introduction

Within the community of people working 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 [3] [4] [6] [7] [13]. 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 [12]. Though both kinds of verification are needed to design complex circuits, these two verification problems have until recently been studied independently and the techniques that have been proposed to deal with them had few relations one with another.

The behavioral equivalence between two machines $\mathcal{M}_1$ and $\mathcal{M}_2$ can be proved by traversing the state-transition graph of the product machine $\mathcal{M}_1 \times \mathcal{M}_2$ [12]. The first techniques that have been proposed were based on a double enumeration of the states and the input patterns of the machines [11], 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 $\mathcal{M}_2$.

More recently we have presented [8] [9] a proof procedure that manipulates sets of states and inputs. The basic concepts that underlie this proof procedure are (1) to represent sets either by their *characteristic functions* or by *vectors of Boolean functions* [9] and (2) that the operations on these Boolean functions can be efficiently performed by denoting them with typed decision graphs [1]. This procedure can handle machines that could not be treated with the enumeration based methods referenced above.

It has been shown [3] [6] that these basic concepts can also be used to develop a procedure for proving that a sequential machine holds some temporal properties without building its state-transition graph. Such procedures could overcome the limits of the methods that have been developed in the past, which all required the partial or total building of this graph. However the procedure described in [3] [6] requires the construction of the *transition relation* of the sequential machine, which is not feasible for most of complex machines.

This paper presents the proof procedure that we have independently developed to check that a machine holds a temporal property. This technique, which does not use the transition relation of the machine, is based on specific operators called "restrict" and "expand". The paper is divided into 4 parts. Part 2 defines the model of machines and the temporal formulas that the proof system handles. Part 3 gives the proof algorithm used by the system and shows that it uses the standard Boolean operations in addition with the computation of a specific term. Part 4 is dedicated to the computation of this term that we call the "critical term" because its evaluation is the bottleneck of the algorithm. Part 5 gives some experimental results and discusses them.

2 Definitions

This section describes the inputs of the verification system presented in this paper. First it defines the model of sequential machines that the system handles. Then it gives the syntax and the semantics of the temporal formulas to be verified.

2.1 Model of Sequential Machines

The sequential machine $\mathcal{M}$ that must be verified is an uncompletely specified deterministic Moore Machine. This machine is defined by the 6-tuple $(n, m, r, \omega, \delta, \text{Init})$, where: the state space of the machine $\mathcal{M}$ is $\{0, 1\}^n$, its input space is $\{0, 1\}^m$, and its output space is $\{0, 1\}^r$; ω is the partial output function from $\{0, 1\}^n \times \{0, 1\}^m$ into $\{0, 1\}_\perp^r$; δ is the partial transition function from $\{0, 1\}^n \times \{0, 1\}^m$ into $\{0, 1\}_\perp^n$; Init is the characteristic function of the set of initial states of $\mathcal{M}$.

In this description, all the Boolean functions are denoted by typed decision graphs (TDG), which are a very compact graphical canonical form [1]. These Boolean functions are automatically computed from the behavioral description of the machine using symbolic execution [2]. The transition function δ is denoted by a couple (Cns, F). Cns denotes the domain where the transition function is defined, so it is a Boolean function from $\{0, 1\}^n \times \{0, 1\}^m$ into $\{0, 1\}$. It is built out of constraints given by the designer and additional constraints computed during the symbolic execution [2]. F is a vectorial function $[f_1 \ldots f_n]$, where each f_j is a Boolean function from $\{0, 1\}^n \times \{0, 1\}^m$ into $\{0, 1\}$. The partial function δ is then defined by:

$$\delta = \lambda s.\lambda p.(\textbf{if } \text{Cns}(s, p) \textbf{ then } F(s, p) \textbf{ else } \perp)$$

For the sake of simplicity, we assume in the following that ω is completely specified. We also assume that $(\forall s\ \exists p\ \text{Cns}(s, p))$ is a tautology, which means that any state has at least one successor. This is not a restriction, because temporal formula have no meaning on states that have no successor. If $(\forall s\ \exists p\ \text{Cns}(s, p))$ is not a tautology, then $\lambda s.(\forall p \neg \text{Cns}(s, p))$ is the characteristic function of the set of states that have no successor.

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 [7]. This logic is a formalism specially developed to express properties about the states and the computation paths of finite state systems. The different kinds of state formulas and their meanings with respect to a state of the machine $\mathcal{M} = (m, n, r, \omega, \delta, \text{Init})$ are the following:

1. $r_1, r_2, \ldots, r_n, o_1, o_2, \ldots, o_r$ are state formulas. For any state s of the machine, $s \models r_j$ if and only if the j-th component of s is 1. For any state s of the machine, $s \models o_j$ if and only if the j-th component of $\omega(s)$ is 1.
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 meanings, for instance $s \models (f \wedge g)$ if and only if $s \models f$ and $s \models g$.
3. If f is a state formula, so are the formulas $EX(f)$ and $AX(f)$. For any state s of $\mathcal{M}$, $s \models EX(f)$ iff there exists at least one input pattern p such that $\delta(s, p) \models f$; by definition, $AX(f)$ is $\neg (EX(\neg f))$.

4. If f and g are state formulas, so are the formulas $E[f\ U\ g]$ and $A[f\ U\ g]$. For any state s of the machine $\mathcal{M}$, $s \models E[f\ U\ g]$ if and only if there exists at least one path $(s_0, s_1, ...)$ with $s_0 = s$, and: $\exists i\ ((s_i \models g) \wedge (\forall j\ (0 \leq j < i \Rightarrow s_j \models f)))$.
For any state s of the machine $\mathcal{M}$, $s \models A[f\ U\ g]$ if and only if for all paths $(s_0, s_1, ...)$ such that $s_0 = s$, we have: $\exists i\ ((s_i \models g) \wedge (\forall j\ (0 \leq j < i \Rightarrow s_j \models f)))$.

Abbreviations have been added to make the CTL formulas more legible: $EF(f) =_{def} E[True\ U f]$; $AF(f) =_{def} A[True\ U f]$; $EG(f) =_{def} \neg\ (AF(\neg f))$; $AG(f) =_{def} \neg\ (EF(\neg f))$, i. e. f holds on any state of every path. The reader can refer to [7] to find examples of properties of finite state systems expressed in CTL.

3 Verification Algorithm

It has been shown that "safety properties" of finite state systems [4] can be checked with the forward symbolic traversal procedure presented in [9]. The machine $\mathcal{M}$ holds a safety property f if and only if some sub-part $\mathcal{M}_f$ of the machine $\mathcal{M}$ has an observable behavior that is equivalent to the one of the most general model of the formula f, which is a finite state automaton $\mathcal{A}_f$. This automaton can be automatically built from the formula f. In this case the proof is made by traversing the state diagram of the machine $\mathcal{M}_f \times \mathcal{A}_f$ without building it [10]. For instance, to prove that the machine $\mathcal{M}$ satisfies the formula $f = AG\ (r_i \Rightarrow AX\ (AG\ (o_j)))$ comes down to proving that the observable behavior of $\mathcal{M}$ at its j-th output is equivalent to the behavior of the automaton $\mathcal{A}_f$ given in Figure 1.

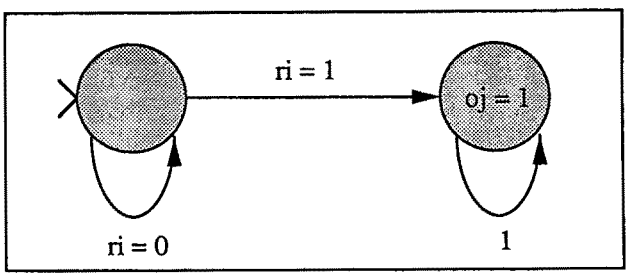

Figure 1. The automaton $\mathcal{A}_f$ associated to the formula $f = AG\ (r_i \Rightarrow AX\ (AG\ (o_j)))$.

This part presents a more general proof algorithm that can handle all the state formulas described in the previous section. This algorithm 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 6-tuple $(m, n, r, \omega, (Cns, F), Init)$ and the formula f to be verified. It recursively computes the set of states $\mathcal{F}$ that satisfy the formula f from the sets of states that satisfy the sub-formulas of f. At each step there are only four basic cases to consider that correspond to the four kinds of formulas given in Section 2.2. Once the set of states $\mathcal{F}$ that satisfy the whole formula f is obtained, to check whether $s \models f$ for some state s of $\mathcal{M}$ comes down to checking whether s belongs to $\mathcal{F}$.

The verification algorithm manipulates sets represented by their characteristic functions, and functions are denoted by their TDG's. We will make no distinction between a function and its TDG, or between a set and the TDG of its characteristic function. The quantifiers that will be used in the sequel are handled using the following identities: if x is a propositional variable, $(\exists x\ f(x))$ is equivalent to $(f(0) \vee f(1))$, and $(\forall x\ f(x))$ is equivalent to $(f(0) \wedge f(1))$. For any n-tuple $x = [x_1 ... x_n]$ of propositional variables, $(Qx\ f(x)) =_{def} (Qx_1 ... Qx_n\ f(x_1, ..., x_n))$, where Q is either the existential or universal quantifier. The size of a TDG f is its number of vertices, and it will be noted $|\ f\ |$. The size of the TDG of a function is relative to a total ordering of its variables [1] [5].

The formulas of type (1) and (2) are trivial to treat. For instance, if $f = (f_1 \wedge f_2)$, and $\mathcal{F}_1$ and $\mathcal{F}_2$ are the characteristic functions of the sets of states that satisfy f_1 and f_2 respectively, then $\mathcal{F}$ is $\lambda s.(\mathcal{F}_1(s) \wedge \mathcal{F}_2(s))$.

The cost of computing the TDG of $\mathcal{F}$ is in $O(|\mathcal{T}_1| \times |\mathcal{T}_2|)$ [1] [5]. As soon as $\mathcal{F}$ is computed, it is trivial to determine whether $s \models f$, since this is equivalent to compute $\mathcal{F}(s)$, which is done in O(n), and then to test whether $\mathcal{F}(s) = 1$. In the same way, $M \models f$ if and only if $\lambda s.(\text{Init}(s) \Rightarrow \mathcal{F}(s)) = 1$, which can be tested in $O(|\text{Init}| \times |\mathcal{T}|)$.

3.1 *EX* and *AX* Formulas

The sets of states that satisfy the formulas of types *EX* and *AX* are computed in a single step. Let f be a formula and $\mathcal{F}$ be the set of states that satisfy f. The set of states $\mathcal{EX}$ of the machine $\mathcal{M}$ that satisfy the temporal formula *EX(f)* is $\{s / \exists p \ \delta(s, p) \in \mathcal{F}\}$. Its characteristic function is:

$$\mathcal{EX} = \lambda s.(\exists p \ \text{Cns}(s, p) \wedge \mathcal{F}(F(s, p))). \tag{1}$$

By definition, $AX(f) = \neg EX(\neg f)$, so the characteristic function $\mathcal{AX}$ of the set of states of the machine $\mathcal{M}$ that satisfy *AX(f)* is:

$$\mathcal{AX} = \lambda s.(\neg (\exists p \ \text{Cns}(s, p) \wedge \neg \mathcal{F}(F(s, p)))). \tag{2}$$

3.2 *EU* and *AU* Formulas

The sets of states that satisfy the formulas of types *EU* and *AU* are computed using iterating algorithms [3]. Let f and g be two formulas and $\mathcal{F}$ and $\mathcal{G}$ be the sets of states that satisfy f and g respectively. The set $\mathcal{EU}$ of states that satisfy the formula *E[f U g]* is the limit of the converging sequence $(\mathcal{E}_j)$ of sets defined by: $\mathcal{E}_0 = \mathcal{G}$, and $\mathcal{E}_{k+1} = \mathcal{E}_k \cup \{s / (s \in \mathcal{F}) \wedge (\exists p \ \delta(s, p) \in \mathcal{E}_k)\}$. The characteristic functions of these sets are the following:

$$\mathcal{E}_0 = \mathcal{G}, \text{ and}$$

$$\mathcal{E}_{k+1} = \lambda s.(\mathcal{E}_k(s) \vee (\mathcal{F}(s) \wedge (\exists p \ \text{Cns}(s, p) \wedge \mathcal{E}_k(F(s, p))))). \tag{3}$$

In the same way the set $\mathcal{AU}$ of states that satisfy the formula *A[f U g]* is the limit of the converging sequence $(\mathcal{A}_j)$ defined by: $\mathcal{A}_0 = \mathcal{G}$, and $\mathcal{A}_{k+1} = \mathcal{A}_k \cup \{s / (s \in \mathcal{F}) \wedge (\forall p \ (\delta(s, p) \neq \perp) \Rightarrow (\delta(s, p) \in \mathcal{A}_k))\}$. The characteristic functions of these sets are the following:

$$\mathcal{A}_0 = \mathcal{G}, \text{ and}$$

$$\mathcal{A}_{k+1} = \lambda s.(\mathcal{A}_k(s) \vee (\mathcal{F}(s) \wedge (\forall p \ \text{Cns}(s, p) \Rightarrow \mathcal{A}_k(F(s, p))))). \tag{4}$$

3.3 The Critical Term

In the formulas (1) to (4) given in the preceding sections two terms appear that actually are the same one, and that has the form $(\exists p \ \text{Cns}(s, p) \wedge \chi(F(s, p)))$. Since the formula $(\forall x \ f)$ is equivalent to $(\neg (\exists x \ \neg f))$, the equation (4) can be rewritten into:

$$\mathcal{A}_{k+1} = \lambda s.(\mathcal{A}_k(s) \vee (\mathcal{F}(s) \wedge \neg (\exists p \ \text{Cns}(s, p) \wedge \neg \mathcal{A}_k(F(s, p))))). \tag{4'}$$

From the computational point of view this means that the four basic cases of CTL formulas can be treated with the standard Boolean operations (negation, conjunction, and disjunction), the test of equivalence (to test whether the sequences $\mathcal{E}_k$ and $\mathcal{A}_k$ have converged), in addition with the evaluation of the *critical term* $(\exists p \ \text{Cns}(s, p) \wedge \chi(F(s, p)))$, where χ is a TDG denoting a Boolean function from $\{0, 1\}^n$ into $\{0, 1\}$. Next section explains why we call this term the "critical term", and discusses how it can be computed.

4 Computing the Critical Term

This section shows that computing the critical term is the bottleneck in the verification algorithm. Then it explains why the critical term can be easily computed when the TDG of the transition relation of the machine can be build [3] [6]. At last, it presents the techniques we have developed to perform this computation when it is not possible to build this TDG, which happens for most of complex circuits.

4.1 Is the Computation of the Critical Term a Difficult Problem?

Since typed decision graphs are canonical, testing the equivalence of two TDG's is in O(1). All the standard Boolean operations on TDG's are performed in polynomial time [1] [5]. This means that the computational cost of the formal verification of CTL formulas depends on the evaluation of the critical term.

The complexity of the critical term computation can be studied using the composition problem. This problem is to compute the TDG of $\lambda x.g(f_1(x), ..., f_n(x))$, from the TDG's of the Boolean functions $f_1, ..., f_n$, and g. The three following theorems show that computing the critical term is the most difficult operation that appears in the algorithm presented in Section 3.

Theorem 1. *Composition is O(1)-reducible to the computation of the critical term.*

Proof. It is immediate that composition is reducible in O(1) to the evaluation of the critical term $(\exists p\ Cns(s, p) \wedge \chi(F(s, p)))$, where $Cns = 1$, $\chi = g$, and $\lambda s.\lambda p.F = [\lambda s.\lambda p.f_1(s)\ ...\ \lambda s.\lambda p.f_n(s)]$. ❑

Theorem 2. *Composition is NP-hard.*

Proof. Let $C = \bigwedge_j c_j$ be a 3-conjunctive normal form made of n clauses c_j. Each clause c_j is a disjunction $(q_{1j} \vee q_{2j} \vee q_{3j})$ of three literals, so the TDG's of these clauses are built in O(n). Let g be the function $\lambda[y_1 ... y_n].(\bigwedge_j y_j)$, whose TDG is a n-vine built in O(n). Since C is satisfiable if and only if the TDG of $g(c_1, ..., c_n)$ is not equal to 0, which can be tested in O(1), composition is NP-hard. ❑

The two precedings theorems show that the computation of the critical term is NP-hard. By using the hypothesis that the order of the variables cannot be modified, we obtain the more precise following theorem.

Theorem 3. *Composition is a no polynomial problem if the ordering of the variables is fixed. More precisely, there exist some TDG's $f_1, ..., f_n$ and g, using O(n) variables, with $\Sigma_j |f_j| + |g| = O(n)$, and such that $|g(f_1, ..., f_n)| = O(2^n)$.*

Proof. We consider 2n variables $y_1 < ... < y_{2n}$, and the function $g = \lambda[x_1 ... x_n].(\bigwedge_j x_j)$. Its TDG is a n-vine of size n. We define n functions $f_j = \lambda[y_1 ... y_{2n}].(y_j \Leftrightarrow y_{2n-j+1})$. The TDG's of the functions f_j have all two vertices, so $\Sigma_j |f_j| = 2n$. Thus $|g| + \Sigma_j |f_j| = O(n)$. The composition $g(f_1, ..., f_n)$ is the function $\lambda y.(\bigwedge_j f_j(y))$, that is $\lambda[y_1 ... y_{2n}].(\bigwedge_j (y_j \Leftrightarrow y_{2n-j+1}))$, but the TDG of the term $((y_1 \Leftrightarrow y_{2n}) \wedge (y_2 \Leftrightarrow y_{2n-1}) \wedge ... \wedge (y_n \Leftrightarrow y_{n+1}))$ with the order $y_1 < ... < y_{2n}$ has a size in $O(2^n)$. ❑

This last theorem shows that computing the critical term on TDG's is at least exponential in the worst case. However, one can object that the TDG of the term $((y_1 \Leftrightarrow y_{2n}) \wedge (y_2 \Leftrightarrow y_{2n-1}) \wedge ... \wedge (y_n \Leftrightarrow y_{n+1}))$ is in O(n) with the variable ordering $y_1 < y_{2n} < y_2 < y_{2n-1} < ... < y_n < y_{n+1}$. We may think that composition can be computed more efficiently if we are able to find a variable ordering that minimizes the sizes of the manipulated TDG's. The following remarks show that the problem is not such simple. First, finding such a variable ordering is itself a NP-hard problem. Second, there exist some functions of n variables whose TDG has a no polynomial size with respect to n, whatever the variable ordering. The last remark shows that the composition, and so the computation of the critical term, is *certainly* no polynomial even if an oracle would provide dynamically a "good order".

4.2 Using the Transition Relation to Compute the Critical Term

The transition relation Δ of the machine $\mathcal{M}$ is a subset of $\{0, 1\}^n \times \{0, 1\}^n$. For any states s and s', (s, s') belongs to Δ if and only if $(\exists p\ (s' = \delta(s, p)))$. This means that $\Delta =_{def} \lambda s.\lambda s'(\exists p\ Cns(s, p) \wedge (s' \equiv F(s, p)))$. Using the transition relation Δ, the critical term $(\exists p\ Cns(s, p) \wedge \chi(F(s, p)))$ can be rewritten into:

$$(\exists s'\ \Delta(s, s') \wedge \chi(s')).$$

Given the TDG's of Δ and χ, the TDG of $(\Delta(s, s') \wedge \chi(s'))$ can be built in $O(|\Delta| \times |\chi|)$, and the critical term is obtained by eliminating from this TDG the n atoms associated to s'. Note that this elimination is not polynomial with respect to n in the worst case, but experience shows it remains polynomial on practical examples.

Assuming that the TDG of Δ can be built, this technique is very efficient [6], because it uses only the standard logical operators, which have relatively low computational costs. But building Δ requires the computation of the TDG of $(Cns(s, p) \wedge (\bigwedge_j (s_j' \equiv f_j(s, p))))$, which is done in $O(|Cns| \times 2^n \times \Pi_j |f_j|)$. Experience shows that for complex machines, it is not possible to build the TDG of the transition relation Δ, so others techniques are needed.

4.3 Using the "Restrict" and the "Expand" Operators to Compute the Critical term

The term $(\exists p\ Cns(s, p) \wedge \chi(F(s, p)))$ can be computed in two steps. The first step consists in building the TDG of the formula $(Cns(s, p) \wedge \chi(F(s, p)))$, and the second step in eliminating from this TDG the atoms associated to the input pattern p. The term $\chi(F(s, p))$ is the composition of χ with the vectorial function $F = [f_1 \ldots f_n]$, whose computation is exponential in the worst case, as shown in Section 4.1. More precisely, it can be easily shown, using a proof similar to the one of Theorem 3, that the computation of the TDG $\chi(f_1, \ldots, f_n)$ from the TDG's of $f_1, \ldots, f_n$ and χ is at least in $O(|\chi| \times \Pi_j |f_j|)$.

Section 4.3.1 shows that it is not necessary to build explicitely the TDG of $\chi \circ F$ to compute the term $(\exists p\ Cns(s, p) \wedge \chi(F(s, p)))$. It presents the "expand" operation that avoids this construction. Section 4.3.2 presents the "restrict" operator that further reduces the computational cost of the composition by reducing the sizes of the TDG's f_j used in the term $\chi(F(s, p))$.

4.3.1 The "Expand" Operation

The idea that underlies the "expand" operation is to express the term $\chi(F(s, p))$ as a sum of functions whose TDG's have less vertices than the TDG of $\chi(F(s, p))$. Using these functions noted $h_1, \ldots, h_k$, the critical term can be rewritten into $(\exists p\ Cns(s, p) \wedge (\bigvee_j h_j(s, p)))$, that can be further transformed into $(\exists p \bigvee_j (Cns(s, p) \wedge h_j(s, p)))$ by distributing the conjunction over the disjunction. The existential quantifier commutes with the disjunction, so that the critical term finally becomes:

$$(\bigvee_j (\exists p\ Cns(s, p) \wedge h_j(s, p))).$$

The final form of this term expresses that its computation can be decomposed into a sequence of computation of simpler terms.

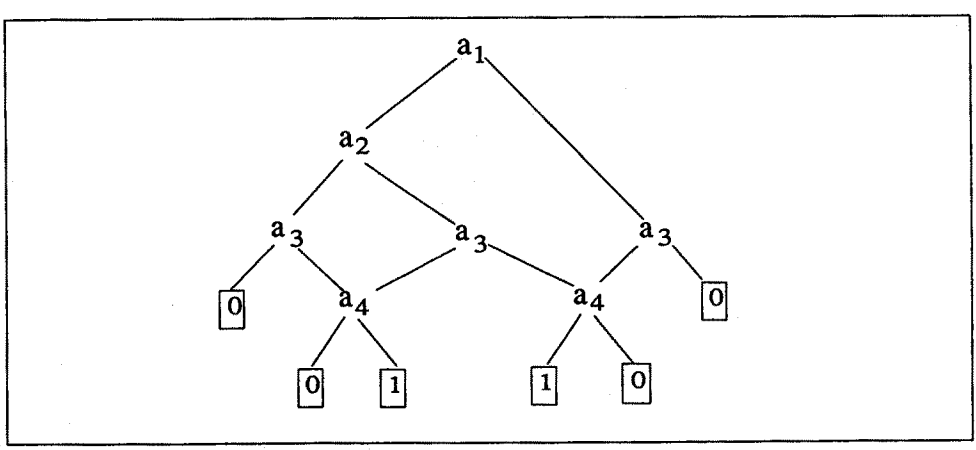

Figure 2. Application of the Expand Function on a BDD.

Consider the functions $f_1, ..., f_n$, and g. The computation of a list of Boolean functions $h_1, h_2, ..., h_k$ whose sum is equal to the function $g(f_1, ..., f_n)$ is made by the "expand" operation. For clarity's sake, we present here the "expand" operation on the binary decision diagrams (BDD) [5].

Each path in the BDD of the function g starting from the root and leading to the leaf 1 defines a *cube* c_j of the function g, and the function g is equal to the sum of its cubes. This means that we could take as functions h_j the functions $c_j(f_1, ..., f_n)$. However this would lead to perform many redundant computations. Consider for instance the BDD drawn in Figure 2. In this BDD there are four paths from the root to 1 that define the four cubes $c_1 = (\neg a_1 \wedge \neg a_2 \wedge a_3 \wedge a_4)$, $c_2 = (\neg a_1 \wedge a_2 \wedge \neg a_3 \wedge a_4)$, $c_3 = (\neg a_1 \wedge a_2 \wedge a_3 \wedge \neg a_4)$, and $c_4 = (a_1 \wedge \neg a_3 \wedge a_4)$. The four functions that would be computed are $h_1 = (\neg f_1 \wedge \neg f_2 \wedge f_3 \wedge f_4)$, $h_2 = (\neg f_1 \wedge f_2 \wedge \neg f_3 \wedge f_4)$, $h_3 = (\neg f_1 \wedge f_2 \wedge f_3 \wedge \neg f_4)$, $h_4 = (f_1 \wedge \neg f_3 \wedge f_4)$.

It is obvious that during the computation of the functions h_1, h_2, h_3, and h_4, the product $(\neg f_1 \wedge f_2)$ is computed twice, as well as the product $(\neg f_3 \wedge f_4)$. Some of these redundant computations can be eliminated by storing partial results in the vertices of the graph that are shared. This is done by the function "expand" that performs a top-down traversal of the BDD of g, and stores in each vertex v of g, the BDD of the function $C_v(f_1, ..., f_n)$, where C_v is the sum of all the cubes represented by the paths starting from the root of the BDD of g and leading to v. Each time the top-down traversal reaches a leaf equal to 1, the function "expand" produces one of the functions h_j.

For the BDD drawn in Figure 2, the partial functions stored in the vertices are: (1) for the vertex (1), $(\neg f_1)$ for (2), $(\neg f_1 \wedge \neg f_2)$ for (3), $(\neg f_1 \wedge f_2)$ for (4), $((\neg f_1 \wedge \neg f_2 \wedge f_3) \vee (\neg f_1 \wedge f_2 \wedge \neg f_3))$ for (5), (f_1) for (6), and $((\neg f_1 \wedge f_2 \wedge f_3) \vee (f_1 \wedge \neg f_3))$ for (7), and the "expand" operation produces two functions:

$$h_1 = (f_4 \wedge ((\neg f_1 \wedge \neg f_2 \wedge f_3) \vee (\neg f_1 \wedge f_2 \wedge \neg f_3))), \text{ and}$$

$$h_2 = (\neg f_4 \wedge ((\neg f_1 \wedge f_2 \wedge f_3) \vee (f_1 \wedge \neg f_3))).$$

Experience shows that the TDG's of the functions $h_1, ..., h_k$ generated by the "expand" operation are smaller than the TDG of the function $\chi(F(s, p))$. The time needed to compute each of these functions directly depends on the sizes of the TDG's of the functions $\delta_j(s, p)$. Next section presents a new Boolean operator that can be used to reduce the sizes of the TDG's used in the term $\chi(F(s, p))$.

4.3.2 The "Restrict" Operator

This section presents the Boolean operator "restrict" [8], noted "$\Downarrow$", that gives a means to reduce the sizes of the TDG's of the functional vector F(s, p) used in the formula $\chi(F(s, p))$. This directly reduces the computational cost of the critical term.

The idea that led to the "$\Downarrow$" operator is made clear by the following remark. In the equation (3) given in Section 3.2:

$$\mathcal{E}_{k+1} = \lambda s.(\mathcal{E}_k(s) \vee (\mathcal{F}(s) \wedge (\exists p \ Cns(s, p) \wedge \mathcal{E}_k(F(s, p))))), \qquad (3)$$

as soon as the term $\mathcal{E}_k(s)$ is equal to 1 or the term $\mathcal{F}(s)$ is equal to 0, the value of $\mathcal{E}_{k+1}(s)$ does not depend on the value of $(\exists p \ Cns(s, p) \wedge \mathcal{E}_k(F(s, p)))$. In the same way, when Cns(s, p) is equal to 0, then so is $(Cns(s, p) \wedge \mathcal{E}_k(F(s, p))$. This means that in the term $\mathcal{E}_k(F(s, p))$, the vectorial function F can be replaced with its restriction $F|_D$ to the domain D, where D is the set whose characteristic function is $\lambda s.\lambda p.(\neg \mathcal{E}_k(s) \wedge \mathcal{F}(s) \wedge Cns(s, p))$.

The same remark holds for the equations (1), (2), and (4'). For these equations the characteristic functions of the domains D to which the vectorial function F can be restricted are the following:

$$D = Cns \quad \text{for (1) and (2),}$$

$$D = \lambda s.\lambda p.(\neg \mathcal{E}_k(s) \wedge \mathcal{F}(s) \wedge Cns(s, p)) \quad \text{for (3),}$$

$$D = \lambda s.\lambda p.(\neg \mathcal{A}_k(s) \wedge \mathcal{F}(s) \wedge Cns(s, p)) \quad \text{for (4').}$$

The "restrict" operator has two arguments f and D that are Boolean functions. Its definition [8] on Shannon's canonical form is given in Figure 3. In this figure, we note f.root the atom that occurs at the top of Shannon's canonical form of f, and (f/¬a, f/a) the Shannon's decomposition of f with respect to the atom a. We have the identity: $f = (\neg a \wedge f/\neg a) \vee (a \wedge f/a)$.

```
function restrict(f, D);
if D = 0 then error;
if D = 1 or f = 0 or f = 1 then return f;
let a = D.root in {
   if D/¬a = 0  then return restrict(f/a, D/a);
   if D/a  = 0  then return restrict(f/¬a, D/¬a);
   if f/¬a = f/a then return restrict(f, D/¬a ∨ D/a);
   return ((¬a ∧ restrict(f/¬a, D/¬a)) ∨ (a ∧ restrict(f/a, D/a)));
   }
```

Figure 3. Semantics of the Restrict Operator.

The main properties of "$\Downarrow$" are expressed by the following theorems. Their proofs are made by induction on Shannon's canonical form of f and D, and by case analysis.

Theorem 4. *Let f and $D \neq 0$ be two Boolean functions. Then $D(x) = 1$ implies that $(f \Downarrow D)(x) = f(x)$.*

Theorem 5. *Let f and D be two Boolean functions, with $D \neq 0$. If $(D \Rightarrow f)$, then $(f \Downarrow D) = 1$. If $(D \Rightarrow \neg f)$, then $(f \Downarrow D) = 0$.*

Theorem 6. *Let f and D be two Boolean functions, with $D \neq 0$. Amongst the functions g such that $(D \Rightarrow (g \equiv f))$ is a tautology, there is a unique function whose Shannon's canonical form has a minimal number of vertices, and this function is $(f \Downarrow D)$.*

Corollary. *Let f and D be two Boolean functions, with $D \neq 0$. Shannon's canonical form of $(f \Downarrow D)$ has less or the same number of vertices than Shannon's canonical form of f.*

Remark. Each time the case (D/¬a = 0) or (D/a = 0) is used to recursively compute Shannon's canonical form of (f $\Downarrow$ D), one branch of Shannon's canonical form of f is deleted. This means that the sooner a "0" occurs on a branch of Shannon's canonical form of D, the larger the number of vertices of Shannon's canonical form of f that are deleted is. This expresses that the smaller the set D is, the greater the reduction is [8].

The operator "$\Downarrow$" defined above on Shannon's canonical form can also be defined on TDG's. Passing from the canonical tree representation to the canonical graph representation gives rises to the following problem. It can happen that during the application of the operator "$\Downarrow$" to the TDG's of the function f and of D, some vertex that is shared in the TDG of the function f gets restricted several times, with different vertices from D. This can lead to create a TDG that has more vertices than the one of f. This means that Theorems 4 and 5 hold for this operator on TDG's, but not Theorem 6. To assure that the application of the operator "$\Downarrow$" on the TGD's of f and D does not return a larger TDG than the one of f, it is sufficient to compare the sizes of the TDG's of f and of (f $\Downarrow$ D), and to return the smallest TDG, which is done in $O(\max(|f|, |D|))$. The function that returns the TDG of (f $\Downarrow$ D) uses Theorem 5 to speed up the computation, and uses a cache to avoid redundant computations. The complexity of the operator "$\Downarrow$" is $O(|f| \times |D|)$.

4.4 Discussion of the Method

The computation of the critical term depends mainly on the cost of the composition. The "expand" operation can be basically seen as a method to compute $\chi(f_1, ..., f_n)$, whose computational cost is at least

$O(|\chi| \times \Pi_j |f_j|)$. Theorem 4 assures that, for the domains D associated to the equations given in Section 4.3.2, the application of the operator "$\Downarrow$" on each component of the vectorial function $[f_1 \ldots f_n]$ is valid. Applying the "$\Downarrow$" operator on F with the domain D is made in $O(|F| \times |D|)$. The TGD's of the functions $(f_1 \Downarrow D), \ldots, (f_n \Downarrow D)$ have less vertices than the ones of the functions $f_1, \ldots, f_n$. Assuming that each TDG f_j has a size s, and that each TDG $(f_j \Downarrow D)$ has a size bound by $(s \times R)$, where $0 \leq R \leq 1$ is the reduction ratio, the composition $\chi \circ (F \Downarrow D)$ is made with a gain of complexity of at least $(1 / R)^n$. This means that the quadratic operation "$\Downarrow$" can reduce exponentially the computational cost of the critical term.

5 Experimental Results - Discussion

The verification algorithms given above have been written in LISP, and the CPU times given here have been obtained on a BULL DPX5000 mini computer with 32 megabytes of main memory.

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 has more than 10000 vertices and the largest of them has more than 2500 vertices. This machine has about 377000 valid states out of the 2^{33} possible states. The CPU time needed to symbolically traverse the state diagram of this machine using the procedure given in [9] is 1227 seconds. We did not succeed in computing the TDG of the transition relation of this machine, so the method of [3] and [6] cannot be applied. The property to be proved valid required the computation of only one fixed point that was obtained in 38 steps that took 4000 seconds of CPU time. The composition $\chi(F(s, p))$ of the critical term stopped to be computable using the standard composition algorithm after a very small number of steps. The "restrict" operation was very useful since it reduced the TDG's of $f_1, \ldots, f_{33}$ used in the "expand" operation in such a way that during the iteration, none of them had more than 186 vertices. The largest TDG representing one of the sets in the sequence had 352 vertices.

The sequential circuit *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* is equal to 1 in any valid state of the machine. This property is expressed by the formula $Init \models AG(OK)$. 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 property *AG(OK)* is a safety property. It can be checked by traversing the state diagram of the machine *Sync* using the symbolic traversal procedure given in [9]. At each step during the traversal that starts from the initial state Init of the machine, it is sufficient to check that the output *OK* 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 140 seconds, and the traversal is done in 20 steps. This example seems to show that whenever the formula to be proved is a safety property, it is better to use this procedure rather than the general procedure described in this paper.

In fact, the technique proposed here cannot give to the symbolic backward traversal procedure the performance of the forward symbolic traversal procedure [9]. The symbolic backward traversal procedure is essentially based on the computation of the reverse image of the vectorial function δ on a set χ. The symbolic forward traversal procedure is based on the computation of the image of the vectorial function δ on a set χ. Given a machine specified by the 6-tuple $(n, m, r, \omega, \delta, Init)$, we can show that the reverse image symbolic computation is *intrinsically* more difficult than the image symbolic computation. It is easy to show that image symbolic computation is NP-hard (for instance, composition is linearly reducible to image symbolic computation), and that it is $O(n)$-reducible to reverse image symbolic computation. The problem is that reverse image symbolic computation is not polynomially reducible to image symbolic computation, because reducing the former to the latter requires to inverse the vectorial function δ, which is a NP-hard problem (image symbolic computation itself is O(1)-reducible to vectorial Boolean function inversion).

Conclusion

This paper has presented a proof algorithm that automatically checks whether some uncompletely defined deterministic Moore machine $\mathcal{M}$ holds some property expressed in the CTL formalism. This algorithm does not require the building of the state-transition graph of the machine $\mathcal{M}$, so it overcomes the limits of the previous methods based on this construction. This means that sequential machine with a very large number of states and transitions can be handle with this method.

Moreover, this proof algorithm does not require the building of the TDG of the transition relation of the machine, which is too large for many practical examples. The proof algorithm is based on the algorithms that were initially developed for proving the equivalence between two machines, in addition with a procedure that essentially computes the reverse image of a vectorial Boolean function. This procedure is the bottleneck of the proof algorithm.

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 Ed., 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 Int. Workshop, Applied Formal Methods for Correct VLSI Design*, Leuven, Nov. 1989.

[4] A. Bouajjani, J. C. Fernandez, N. Halbwachs, "An Executable Temporal Logic for Expressing Safety Properties", July 1990.

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

[6] S. Burch, E. M. Clarke, K. L. McMillan, "Sequential Circuit Verification Using Symbolic Model Checking", in *Proc. of Design Automation Conference (DAC)*, Orlando FL, USA, July 1990.

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

[8] O. Coudert, C. Berthet, J. C. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution", in *Lecture Notes in Computer Science: Automatic Verification Methods for Finite State Systems*, Volume 407, J. Sifakis Editor, Springer-Verlag, pp 365-373, June 1989.

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

[10] O. Coudert, C. Berthet, J. C. Madre, "Formal Boolean Manipulations for the Verification of Sequential Machines", in *Proc. of the First European Design Automation Conference (EDAC)*, Glasgow, March 1990.

[11] G. J. Holtzman, "Algorithms for Automated Protocol Validation", in *Lecture Notes in Computer Science: Automatic Verification Methods for Finite State Systems*, Volume 407, J. Sifakis Editor, Springer-Verlag, June 1989.

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

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