A Unified Framework for
the Formal Verification of Sequential Circuits

Olivier Coudert

Jean Christophe Madre

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

1 Introduction

Hardware description languages (HDLs) dramatically change the way circuit designers work. These languages can be used to describe circuits at a very *high level of abstraction,* which allows the designers to specify the behavior of a circuit before realizing it. The validation of these *specifications* is currently done by executing them, which is very costly [2]. This cost motivates the research [3] [5] [7] [10] done on the automatic verification of temporal properties of finite state machines.

Once the design of the circuit is done, the problem is to verify that the resulting circuit is correct with respect to its specification. Until recently, this verification was done by simulating the circuit and its specification on the same input sequences and by comparing their output sequences. This verification method is very costly and incomplete because of the large number of input sequences to consider [2].

This paper presents a unified framework for the verification of synchronous circuits. Within this framework the two verification tasks presented above can be automatically performed using algorithms based on the same concepts. The first idea is to manipulate *sets of states* and *sets of transitions* instead of *individual states* and *individual transitions.* The second idea is to represent these sets by *Boolean functions* and to replace operations on sets with operations on Boolean functions.

Part 2 of the paper defines the two problems addressed here, and then it presents the verification algorithms. It shows that these algorithms use the standard set operations in addition to two specific operations called "Pre" and "Img". Part 3 briefly explains why the basic set operations are very efficiently performed when sets are denoted by the typed decision graphs of their characteristic functions. Part 4 presents the new Boolean operators "Constrain" and "Restrict", and the function "Expand" that support efficiently the "Img" and "Pre" operations. Part 5 gives experimental results and discusses them.

2 The Verification Algorithms

This section defines the model of sequential circuits that will be verified, and the two verification problems addressed here. Then it gives for both problems an algorithm based on set manipulations.

2.1 Definitions

For the sake of clarity we will consider in this paper that the sequential circuits that must be verified are deterministic Moore machines. Moreover we assume that these machines are *completely* specified, which means that for any state of the machine, (1) the outputs are defined, and (2) for any input pattern, the next state of the machine is defined. This is not a limitation since, if the machine is incompletely specified, it is possible to add a dummy state in order to obtain a completely specified machine.

A deterministic Moore machine $\mathcal{M}$ is defined by a 6-tuple $(Y, I, O, \lambda, \delta, \text{Init})$. Y is the vector $[y_1, ..., y_m]$ of Boolean state variables of the machine: a *state* of $\mathcal{M}$ is defined by the Boolean values of the variables $y_1, ..., y_m$. I is the vector of n boolean inputs of the machine. O is the vector of k boolean outputs of the machine. The output function λ is a vector of k Boolean functions (one for each output) from the set $\{0, 1\}^m$ into $\{0, 1\}$. The transition function δ is a vector of m Boolean functions from $\{0, 1\}^m \times \{0, 1\}^n$ into $\{0, 1\}$. Finally Init is the initial state of the machine.

The 6-tuple that defines a sequential circuit can be obtained either from its gate level description or from its functional description, using a symbolic execution process such as the one used in PRIAM [2].

2.2 Verification of Temporal Properties

The temporal formulas that the verification system takes as input are the state formulas of the computation tree logic CTL [7]. This logic is a formalism that was specifically developed to express properties of 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 here defined by the values of its state variables. The 4 basic kinds of CTL state formulas are the following:

(1) $(y_1), ..., (y_n)$ are state formulas. For any state s, $s \models y_j$ if and only if (iff) the value of the variable y_j is 1 in the state s.

(2) If f and g are state formulas then so are the formulas $(\neg f)$, $(f \wedge g)$, $(f \vee g)$, $(f \Leftrightarrow g)$, and $(f \Rightarrow g)$.

(3) If f is a state formula, so are the formulas $EX(f)$ and $AX(f)$: $s \models EX(f)$ iff there exists at least one input pattern p such that $\delta(s, p) \models f$, and $AX(f) =_{\text{def}} \neg EX(\neg f)$.

(4) If f and g are state formulas, so are the formulas $E[f U g]$ and $A[f U g]$: $s \models E[f U g]$ iff there exists at least one path $(s_0, s_1, ...)$ with $s_0 = s$, such that $\exists i \, ((s_i \models g) \wedge (\forall j \, (0 \le j < i \Rightarrow s_j \models f)))$, and $s \models A[f U g]$ iff for all paths $(s_0, s_1, ...)$ such that $s_0 = s$, then $\exists i \, ((s_i \models g) \wedge (\forall j \, (0 \le j < i \Rightarrow s_j \models f)))$.

The first algorithms that have been proposed to verify automatically that some machine holds a temporal property [7] used traversal techniques of its state-transition graph, which had to be partially or entirely built. This limited the application of these algorithms to relatively small machines.

The verification algorithm used here takes as inputs a machine $\mathcal{M} = (Y, I, O, \lambda, \delta, \text{Init})$ and the temporal formula f to be verified. It recursively computes the set of states of $\mathcal{M}$ that satisfy the formula f from the sets of states that satisfy its subformulas. At each step there are only 4 basic cases to consider that correspond to the 4 basic kinds of formulas given above. Once the set of states F that satisfy the whole formula is obtained, to check whether $(s \models f)$ for some state s of $\mathcal{M}$ comes down to checking whether s belongs to F [3].

The sets of states that satisfy formulas of type (1) and (2) can be computed using the basic set operations. For instance, the set of states that satisfy the formula (y_j) is $\{1\} \times \{0, 1\}^{m-1}$; if F and G are the sets of states that satisfy the formulas f and g respectively, then the set of states that satisfy the formula $(f \vee g)$ is $(F \cup G)$.

The other kinds of formulas are treated with the "Pre" operation, either in one step (EX and AX formulas) or by fixed point algorithms (EU and AU formulas). By definition,

$$\text{Pre}(Q, A, B) = \{s / (s \in A) \wedge (Qp \, \delta(s, p) \in B)\},$$

where Q is either the existential "∃" or the universal "∀" quantifier. Pre(Q, A, B) is the subset of states of A which have either at least one successor (Q = ∃) or all their successors (Q = ∀) in the set B. Let f be a formula and F be the set of states that satisfy f. The sets of states EX and AX that satisfy EX(f) and AX(f) respectively are defined by:

$$EX = \text{Pre}(\exists, \{0, 1\}^m, F) \tag{1}$$

$$AX = \text{Pre}(\forall, \{0, 1\}^m, F) \tag{2}$$

Let f and g be two formulas and F and G be the sets of states that satisfy f and g respectively. The sets of states EU and AU that satisfy the formulas E[f U g] and A[f U g] respectively are the limits of the following converging sequences of sets (E_k) and (A_k) [3]:

$$E_0 = G, \text{ and } E_{k+1} = E_k \cup \text{Pre}(\exists, F, E_k), \tag{3}$$

$$A_0 = G, \text{ and } A_{k+1} = A_k \cup \text{Pre}(\forall, F, A_k). \tag{4}$$

These algorithms use only the basic set operations $\cup$, $\cap$, $=$, in addition with the "Pre" operation.

2.3 Comparison of Sequential Machines

The fundamental method for comparing the observable behaviors of $\mathcal{M}_1 = (Y_1, I, O, \delta_1, \lambda_1, \text{Init}_1)$ and $\mathcal{M}_2 = (Y_2, I, O, \delta_2, \lambda_2, \text{Init}_2)$ is to check that the output ok of the *product machine* $\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2$ is equal to 1 for every valid state of $\mathcal{M}$. This machine is defined by $\mathcal{M} = (Y, I, [\text{ok}], \delta, \lambda, \text{Init})$ where $Y = Y_1 @ Y_2$, ("@" is the vector concatenation), $\delta = \delta_1 @ \delta_2$, $\text{Init} = \text{Init}_1 \times \text{Init}_2$ and λ_{ok} depends on the comparison criterion.

The correctness property given above holds for the machine $\mathcal{M}$ *iff* the CTL state formula $(\neg E(\text{True } U (\text{ok} = 0)))$ holds in its initial state Init, so the verification algorithm presented in 2.2 can be used to compare two machines. We give here a specific comparison algorithm that has been shown by experience to be much more efficient than this general algorithm. Both methods will be discussed in Part 5.

The idea used here is to compute the set Valid of all the valid states of the machine $\mathcal{M}$. This set is the limit of the converging sequence of sets V_k defined by the equations:

$$V_0 = \text{Init, and } V_{k+1} = V_k \cup \text{Img}(\delta, V_k \times \{0, 1\}^n), \tag{5}$$

where $\text{Img}(f, A) =_{\text{def}} \{f(a) / a \in A\}$ is the image of the set A with respect to the function f. Once Valid is computed, the verification comes down to testing whether $\text{Img}(\lambda_{ok}, \text{Valid}) = \{1\}$ (6). This algorithm, like the one given in the previous section, uses only the basic set operations, in addition to the "Img" operation.

3 Boolean Functions and Sets

Any subset A of $\{0, 1\}^n$ can be represented by a unique Boolean function χ_A from $\{0, 1\}^n$ to $\{0, 1\}$, defined by: $\chi_A(a) = 1$ if and only if $a \in A$. The function χ_A is called the *characteristic function* of A. The set operators $(\cup, \cap, \subset, \in, \times)$ can be expressed in terms of the logical operators $(\vee, \wedge, \Rightarrow, \neg, \Leftrightarrow)$. For instance, the characteristic function of the set $A \cup B$ is $\lambda y.(\chi_A(y) \vee \chi_B(y))$.

Typed decision graphs [1] are a compact canonical representation of Boolean functions. They have remarkable properties that make the symbolic manipulations on Boolean functions very efficient. Typed decision graphs, which are binary decision diagrams [4] with typed

edges, are the canonical graph representation associated to Shannon's typed canonical form [1]. By associating a unique atom to each of the component of the cartesian product $\{0, 1\}^n$, any characteristic function can be represented by a unique typed decision graph.

The correspondence mentioned above gives the computational cost of the elementary set operations. The negation on typed decision graphs has a null cost so this is the same for the set complementation. The others Boolean operations have a complexity in $O(|G_1| \times |G_2|)$, where $|G|$ is the number of vertices in the graph G, which gives the computational cost of the corresponding elementary set operations.

There is no relation between the number of elements in a set and the number of vertices in the graph of its characteristic function. However there exists some subsets of $\{0, 1\}^n$ whose graphs have $O(2^n / \log(n))$ vertices. Experience shows that, for most of the machines we deal with, while the sets manipulated by the verification algorithms are very large, the graphs of their characteristic functions stay small. This means that the computational cost of the basic set operations performed by the symbolic verification algorithms is low, and that the total cost of these algorithms depends on the costs of the operations "Pre" and "Img".

4 The "Pre" and "Img" Operations

This part explains how the operations "Pre" and "Img" can be easily realized when the typed decision graph of the transition relation and of the output relation of the machine can be build [5]. Then it presents the techniques we have developed to perform these operations when it is not possible to built these graphs, which happens for most complex circuits.

4.1 Using the transition and the output relations

The transition relation Δ of the machine $\mathcal{M}$ is a subset of $\{0, 1\}^m \times \{0, 1\}^n \times \{0, 1\}^m$. For any states s and s' of $\mathcal{M}$, and for any input pattern p, (s, p, s') belongs to Δ if and only if $s' = \delta(s, p)$. For any subset A and B of $\{0, 1\}^m$, the characteristic function of the set Pre(Q, A, B) is equal to:

$$\chi_{\text{Pre}(Q, A, B)} = \lambda s.(\chi_A(s) \wedge (Qp \, \exists s' \, \chi_B(s') \wedge \chi_\Delta(s, p, s'))),$$

The output relation of the machine $\mathcal{M}$, noted Λ, is a subset of $\{0, 1\}^m \times \{0, 1\}^k$. For any state s of $\mathcal{M}$, and for any output pattern out, (s, out) belongs to Λ if and only if out $= \lambda(s)$. The equations (5) and (6), using the operation "Img", can be written as:

$$\chi_{\text{Img}(\delta, A \times \{0, 1\}^n)} = \lambda s'.(\exists p \, \exists s \, \chi_A(s) \wedge \chi_\Delta(s, p, s')) \tag{5'}$$

$$\chi_{\text{Img}(\lambda, A)} = \lambda \text{out}.(\exists s \, \chi_A(s) \wedge \chi_\Lambda(s, \text{out})). \tag{6'}$$

Since the formula $(\exists x \, f(x))$ is equivalent to $(f(0) \vee f(1))$, and the formula $(\forall x \, f(x))$ is equivalent to $(f(0) \wedge f(1))$, the graphs of the characteristic functions of Pre(Q, A, B), $\text{Img}(\delta, A \times \{0, 1\}^n)$ and $\text{Img}(\lambda, A)$ can be directly computed from the graphs of $\chi_A, \chi_B, \chi_\Delta$, and χ_Λ by eliminating the quantified atoms associated with p, s and s'. This technique is very efficient [5], because it uses only the operators $\vee$ and $\wedge$, which have been shown in Section 3 to have relatively low computational costs. The problem is that, for complex machines, it is not possible to build the graphs of Δ and Λ [8].

4.2 The "Restrict" and "Expand" Operators

The equations that define the "Pre" operation are the following:

$$\chi_{\text{Pre}(\exists, A, B)} = \lambda s.(\chi_A(s) \wedge (\exists p \, \chi_B(\delta(s, p)))), \text{ and} \tag{7}$$

$$\chi_{\text{Pre}(\forall, A, B)} = \lambda s.(\chi_A(s) \wedge \neg (\exists p \neg \chi_B(\delta(s, p)))). \tag{8}$$

The graph of the function $\lambda s.(\exists p \, \chi(\delta(s, p)))$, where χ is χ_B or $\neg \chi_B$, can be computed in two steps. The first step consists in computing the function $\chi \circ \delta$. Then, the quantified atoms associated to the input pattern p are eliminated from the graph of this function.

It has been shown [4] that substituting some variable v in the graph G_1 with the graph G_2 has a computational cost in $O(|G_1|^2 \times |G_2|)$. In

order to compute the graph of $\lambda s.\lambda p.(\chi(\delta_1(s, p), ..., \delta_m(s, p)))$, this basic substitution process must be iterated so that all variables of χ are substituted by the graphs of the corresponding components of δ. The problem is that during this composition, some intermediate graphs can be too large to be built.

Section 4.2.1 shows that it is not necessary to build the graph of $\chi \circ \delta$ to compute Pre(Q, A, B). It presents the function "Expand" that avoids this construction. Section 4.2.2 presents the "Restrict" operator that further reduces the computational cost of the "Pre" operation by reducing the sizes of the graphs that are manipulated.

4.2.1 The function "Expand"

The idea that underlies the function "Expand" is to express the function $\chi \circ \delta$ as a sum of K functions $h_1, ..., h_K$, whose graphs have less vertices than the graph of $\chi \circ \delta$. Using these functions, the term $(\exists p\ \chi(\delta(s, p)))$ can be rewritten into $(\vee_j (\exists p\ h_j(s, p)))$. This identity allows us to eliminate directly the quantified atoms associated to the input pattern p from the graphs of the functions $h_1, ..., h_K$, and so the graph of $\chi \circ \delta$ does not have to be built.

Each path in the graph G of the function χ starting from the root and leading to the leaf 1 defines a *cube* c_j of the function χ. This means that the function $c_j \circ \delta$ can be taken as one function h_j. The problems are that the function χ can have $O(2^{|G|})$ cubes, and that even if its number of cubes is relatively small, many redundant computations will be made.

The function Expand performs a top-down traversal of the graph G, and stores in each of its vertices the graph of the function $C_v \circ \delta$, where the function C_v is the sum of all the cubes represented by the paths starting from the root of G and leading to v. Each time the top-down traversal reaches the leaf 1, the function Expand produces one of the functions h_j. The function C_v associated to a vertex v is recursively computed using the functions C_w of the vertices w that point to v. Thanks to the sharing in the graph, partial results are factorized and redundant computations are avoided [10].

Experience shows that the graphs of the functions $h_1, ..., h_K$ generated by the Expand operation are smaller than the graph of $\chi \circ \delta$. The time needed to compute each of these functions directly depends on the sizes of the graphs of the functions δ_j. The next section presents a Boolean operator that can be used to reduce the sizes of the graphs used in the term $\chi(\delta(s, p))$.

4.2.2 The Operator "⇓"

In the equations (7) and (8), when $\chi_A(s) = 0$ the characteristic function of Pre(Q, A, B) is also equal to 0. This means that, in the term $\chi_B(\delta(s, p))$ that occurs in the equations (7) and (8), the transition function δ can be replaced with its restriction to the domain A.

The "Restrict" operator, noted "⇓", takes as input the typed decision graph of a boolean function f and of the characteristic function c of the set to which the function f must be restricted. The semantics of the Restrict operator [8], is given on Shannon's canonical form in Figure 1. In this figure, c.root is the root of Shannon's canonical form of c, and $(c/\neg a, c/a)$ is Shannon's expansion [1] of the function c with respect to a. The main properties of the Restrict operator [8] [10] are expressed by the following theorems.

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

Figure 1. The Restrict Operator on Shannon's canonical form.

Theorem 1. *For any Boolean functions f and $c \neq 0$, if $c(x) = 1$ then* $(f \Downarrow c)(x) = f(x)$.

Theorem 2. *For any Boolean functions f and $c \neq 0$, Shannon's typed canonical form of $(f \Downarrow c)$ has at most the same number of vertices as that of f.*

Theorem 2 is not true for typed decision graphs. It can happen that the graph of $(f \Downarrow c)$ has more vertices than the graph of f. In this case the function restrict returns the graph of f. Experience shows that this case occurs very rarely.

4.3 Performing the "Img" Operation

The problem addressed here is to compute the characteristic function χ of the image of the restriction of a vectorial Boolean function $F = [f_1 ... f_n]$ to a domain defined by its characteristic function χ_A. A definition of χ is [8]:

$$\chi = \lambda[y_1 ... y_n].(\exists x\ \chi_A(x) \wedge (\wedge_j (y_j \Leftrightarrow f_j(x)))).$$

The term $(\wedge_j (y_j \Leftrightarrow f_j(x)))$ represents the transition relation of the machine, which has been shown to be in many cases too complex to be computed [8]. This section presents two algorithms that can be used to compute χ without computing this term.

Both algorithms are based on the "constrain" operator, noted "↓", and work in two steps [8]. The first step common to both algorithms consists in computing a new vectorial function $F' = [f_1' ... f_n']$ such that $Img(F, \chi_A) = Img(F', 1)$. The second step then consists in computing the characteristic function of $Img(F', 1)$, by using co-domain partitioning in the first algorithm and domain partitioning in the second algorithm.

Figure 2 gives the semantics of the operator "↓" [9] on Shannon's canonical form. The operator applies on Shannon's canonical forms of the Boolean functions f and c, and produces Shannon's canonical form of the function $(f \downarrow c)$. Its fundamental properties are expressed by the following theorem [9].

Theorem 3. *Let $F = [f_1 ... f_n]$ be a vector of functions, and c be a function different from 0. Let $F \downarrow c =_{def} [(f_1 \downarrow c) ... (f_n \downarrow c)]$. Then $Img(F \downarrow c, 1) = Img(F, c)$.*

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

Figure 2. The Constrain Operator on Shannon's canonical form.

4.3.1 Co-domain Partitioning Based Algorithm

The first recursive algorithm that computes $Img(F, 1)$ uses the operator "↓" to partition the co-domain of the vectorial function F. The algorithm is a direct application of the following theorem.

Theorem 4. *Let $F_n = [f_1 ... f_n]$ be a Boolean vectorial function. Then: $Img(F_n, 1) = Img(F_{n-1} \downarrow \neg f_n, 1) \times \{0\} \cup Img(F_{n-1} \downarrow f_n, 1) \times \{1\}$.*

The number of recursions needed to compute $Img(F, 1)$ is bound by the number of elements of this set. Several techniques have been proposed to reduce this number of recursions. Vector partitioning [9] consists in splitting the vector F into several sub-vectors of functions which have disjoint supports of variables. In [6] it has been proposed to use a cache where partial results obtained during previous recursions are stored. However the *exact matching* used in [6] can be replaced by an *extended matching* that allows us to match any vector of k components in the cache with $(k! \times 2^k)$ vectors, with a complex-

ity in $O(k \times \log(k))$. This extended matching test is based on the following properties.

Theorem 5. *If χ is the characteristic function of $Img([f_1 ...f_k], A)$, then the characteristic function of $Img([\varepsilon_1(f_1) ... \varepsilon_k(f_k)], A)$, where ε_j is the identity or the negation, is $\lambda[y_1 ... y_n].\chi(\varepsilon_1(y_1), ..., \varepsilon_k(y_k))$.*

Theorem 6. *If χ is the characteristic function of $Img([f_1 ...f_k], A)$, then the characteristic function of $Img([f_{\sigma(1)} ...f_{\sigma(k)}], A)$, where σ is a permutation of the k first integers, is $\lambda[y_1 ... y_k].\chi(y_{\sigma(1)}, ..., y_{\sigma(k)})$.*

4.3.2 Domain Partitioning Based Algorithm

The algorithm based on domain partitioning is a direct application of Theorem 7. The techniques described above can also be used to reduce the number of recursions needed to compute the result. This number is bounded by $\Pi_j |f_j|$, but we think that it is directly related to $|F|$, where $|F|$ is the number of vertices in the graphs that represent F. Note that this algorithm does not create any vertex, except for characteristic functions.

Theorem 7. *Let $F = [f_1 ...f_n]$ be a Boolean vectorial function. Then: $Img([f_1 ...f_n], 1) = Img([f_1/\neg a ...f_n/\neg a], 1) \cup Img([f_1/a ...f_n/a], 1)$.*

5 Experimental Results - Discussion

The algorithms are written in LISP, and the CPU times in seconds are for a BULL DPX5000 mini computer. Figure 3 gives the CPU times needed to compute the set of valid states of some digital circuits. For all circuits, # in is the number of inputs, # reg the number of state variables, depth is the number of iterations, # valid is the number of valid states, t_{codp} and t_{dp} are the CPU times for the algorithm based on co-domain partitioning and for the algorithm based on domain partitioning respectively.

$\mathcal{M}$	# reg	# in	depth	# valid	t_{codp}	t_{dp}
s838	32	35	17	17	2.4	2.5
mclc	11	11	14	35	2.3	2.6
scf	8	27	16	115	6	5.8
s298	14	3	19	218	2.9	2.8
s713	19	35	7	1544	81.7	88.2
s344	15	9	7	2625	32.7	28.6
s382	21	3	151	8865	128	79
s444	21	3	151	8865	126	75.4
cbp16	16	17	2	6.5 E4	1.2	1
cbp32	32	33	2	4.3 E9	5.7	4.5
key	56	62	2	7.2 E16	15.5	4.6
stage	64	113	2	1.8 E19	> 10000	1242
sbc	28	40	10	1.5 E5	> 10000	3530
sync	21	4	20	1469	140	154
clm1	33	13	396	3.8 E5	1227	2620
clm2	32	382	279	3.3 E6	8520	> 10000
mm10	30	13	4	1.8 E8	834	32
mm20	60	23	4	1.9 E17	> 10000	212
mm30	90	33	4	2 E26	> 10000	760

Figure 3. Valid States Computation.

There are circuits that can be treated by only one of the two algorithms. The circuit clm2 can be treated only with co-domain partitioning: at each step during the computation, only a few states are reached. The MinMax [9] circuits (mm20, mm30) can be treated only with domain partitioning: the hit ratio in the cache is very high for this algorithm, which is not the case for the other algorithm, and the number of states that are reached at each step is very large.

The symbolic verification algorithm of temporal properties has been applied to the machines clm1 and Sync. The property to be proved valid on clm1 required the computation of only one fixed point that took 38 steps and 4000s of CPU time to be obtained. The "Restrict" operator was very useful since it reduced the graphs used to compute the term $\chi_B(\delta(s, p))$ in such a way that during the iteration, none of

them had more than 186 vertices, while some of the graphs that represent the transition function of clm1 have more than 2500 vertices. The property to be verified on Sync was Init $\models AG(OK = 1)$. The CPU time needed to make this verification was 4160 seconds and the fixed point was found in 9 steps. The restrict operator was not very useful.

Note that the property $AG(OK = 1)$ can be verified using the algorithm presented in Section 2.3 [10], and the verification times is then the time needed to compute the valid states of Sync, which is very much smaller. This is quite understandable since the "Pre" operation is intrinsically more complex than the "Img" operation [10].

6 Conclusion

In this paper we have shown that the two kinds of verification that are needed to design correct sequential circuits can be treated in a unified framework. We have presented verification algorithms that manipulate sets of states and sets of inputs represented by the typed decision graphs of their characteristic functions.

Though these algorithms are very efficient they do not allow us to deal directly with the complex circuits designed at BULL. This means that some techniques are still needed to exploit the full power of this kernel. These techniques include the use of abstraction mechanisms for separating control and data, and the decomposition of the verification task according to the circuit structure.

References

[1] J. P. Billon, "Perfect Normal Forms for Discrete Functions", *BULL Research Report N°87019*, March 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, November 1989.

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

[5] S. Burch, E. M. Clarke, K. L. McMillan, "Symbolic Model Checking: 10^{20} States and Beyond", in *Proc. of LICS*, 1990.

[6] H. Cho, G. Hachtel, S. W. Jeong, B. Plessier, E. Schwarz, F. Somenzi, "ATPG Aspect of FSM Verification", in *Proc. of ICCAD*, Santa-Clara, U.S.A., June 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 *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989.

[9] 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, November 1989.

[10] O. Coudert, J. C. Madre, C. Berthet, "Verifying Temporal Properties of Sequential Machines without Building their State Diagrams", to appear in *Proc. of the Workshop on Computer Aided Verification*, Rutgers, U.S.A., June 1990.