

Symbolic Computation of the Valid States of a Sequential Machine: Algorithms and Discussion

Olivier Coudert Jean Christophe Madre

Bull Corporate Research Center
Rue Jean Jaurès
78340 Les Clayes-sous-bois FRANCE

Abstract

Computing the valid states of a sequential machine is a problem that appears in several verification and synthesis processes. The computation of the image of a vectorial function is the main operation required to solve this problem. The aim of this paper is to show how the image computation can be symbolically performed, and to discuss the complexities and behaviors of several approaches. We propose a parametric algorithm that performs the image computation. Several instances of this algorithm can be obtained according to the results and the heuristics we give to analyze and to improve the computation. Two instances of this algorithm will be compared on practical examples.

1 Introduction

The valid states of a sequential machine are needed in several verification and synthesis problems: comparing uncompletely defined Mealy machines [7, 8, 12, 13, 16], finding minimal differentiating input sequences, automatic test pattern generation [7], minimal reset sequences generation [15], minimization of sequential circuits [1, 14], elimination of state variables and re-encoding [1].

Until recently, valid state computation was done by simulation. This is very costly because the valid states must be found one per

one. Such enumeration based methods are limited by the number of inputs and valid states of the machines.

To overcome this limitation, it is necessary to describe the valid states in intention rather than in extension: instead of treating individual states and individual input patterns, we will manipulate sets of states and input patterns. For this purpose, we represent the sets with Boolean functions and we replace the set operations with Boolean operations.

This paper shows that the previous algorithms [10] we have proposed are nothing but instances of a more general algorithm for computing valid states of a sequential machine. In this manner, new methods based on new heuristics can be discussed and compared.

Section 2 of the paper first gives the definitions and notations that will be used, and recalls an algorithm that computes the valid states. This algorithm uses the standard set operations in addition to the specific operation called “*Img*”, which performs the image computation. Section 3 discusses the image computation problem. It proposes a parametric algorithm that depends on a cover and on an operator called *range restrictor*. It also justifies the choice of the “constrain” operator as a range restrictor. Section 4 proposes some instances of this algorithm. Section 5 gives experimental results and then discusses some other techniques. At any time, we will outline the complexity and the behavior of the algorithms.

2 Symbolic Valid States Computation

This section gives the definitions and the notations we will use in this paper, and presents an algorithm based on set manipulations to compute the valid states of a sequential machine.

2.1 Boolean Sets, Boolean Functions, and Typed Decision Graphs

A Boolean function is a function from $\{0,1\}^p$ into $\{0,1\}$. Boolean functions are noted f, g, c, χ . We note ε the identity or negation function. A variable of the domain $\{0,1\}^m$ (respectively the codomain $\{0,1\}^n$) is noted \vec{x} or $[x_1 \dots x_m]$ (respectively \vec{y} or $[y_1 \dots y_n]$), where x_k and y_k are propositional variables. The couple of functions $(f[x \leftarrow 0], f[x \leftarrow 1])$ is Shannon’s decomposition [5] of the Boolean function f with respect to the variable x . The formula $(\exists \vec{x} f)$ stands

for $(\exists x_1 \exists x_2 \dots \exists x_m f)$, and $(\forall \vec{x} f)$ is equivalent to $\neg (\exists \vec{x} \neg f)$. The formula $(\exists x f)$ is equivalent to $(f[x \leftarrow 0] \vee f[x \leftarrow 1])$.

Boolean functions are represented by Typed Decision Graphs [2] (TDG's), which are an improvement of Binary Decision Diagrams [5] (BDD's). TDG's and BDD's are canonical (modulo a variable ordering) and compact graph representations of Boolean functions. The results given here hold for both graph representations except when specified. No distinction will be made between a Boolean function and its graph. The size of a function f , noted $|f|$, is the number of vertices of its graph. It depends heavily on the variable ordering [5]. For instance, the size of the largest graph among the functions of n variables is in $O(\frac{2^n}{n})$. A good ordering remains an indispensable condition to work on graphs of reasonable sizes. However, the problem of finding an optimal ordering (i.e. an ordering that minimizes the size of f) is NP-hard, thus heuristics [11] must be used.

Any subset A of $\{0, 1\}^m$ can be represented by a unique Boolean function $\chi_A : \{0, 1\}^m \rightarrow \{0, 1\}$, called the characteristic function of A , and defined by $\chi_A(\vec{x}) = 1$ if and only if $\vec{x} \in A$. All set operations become Boolean operations on characteristic functions. For instance, the characteristic function of the set $A - B$ is $\lambda \vec{x}. (\chi_A(\vec{x}) \wedge \neg \chi_B(\vec{x}))$. The computational cost of the elementary set operations when sets are represented by the graphs of their characteristic functions is then easy to define. Since the negation of a TDG f (respectively a BDD f) is in $O(1)$ (respectively in $O(|f|)$), the complexity of set complementation is the same. The other Boolean operations ($\vee, \wedge, \Rightarrow, \Leftrightarrow, \oplus$) on two graphs f and g have a complexity in $O(|f| \times |g|)$, which gives the computational cost of the other elementary set operations ($\cup, \cap, \times, \subseteq, -, \Delta$). There is no relation between the number of elements of a set and the size of its characteristic function, which enables us to manipulate very large sets providing that their characteristic functions have moderate sizes.

We will also use vectorial Boolean functions. The vectors of functions are noted $\vec{f}, \vec{f}', [f_1 \dots f_n]$. Boolean operators are extended on vectors. For example, $\vec{f} \vee \vec{f}' = [(f_1 \vee f'_1) \dots (f_n \vee f'_n)]$, and $g \wedge \vec{f} = [(g \wedge f_1) \dots (g \wedge f_n)]$. We note “@” the concatenation on vectors.

2.2 Sequential Machines and Valid States

The sequential machines we consider here are uncompletely specified Mealy's machines [12]. An uncompletely Mealy's machine \mathcal{M} is defined by a 5-tuple $(n, m, \omega, \delta, Init)$, where:

- n is the number of Boolean state variables of the machine, so the state space of \mathcal{M} is $\{0, 1\}^n$.
- m is the number of Boolean inputs of the machine, so the input space of \mathcal{M} is $\{0, 1\}^m$.
- ω is the uncompletely defined output function of the machine. If \mathcal{M} has r outputs, ω is a function from $\{0, 1\}^n \times \{0, 1\}^m$ into $\{0, 1, \perp\}^r$.
- δ is the uncompletely defined transition function of the machine. It is a function from $\{0, 1\}^n \times \{0, 1\}^m$ into $\{0, 1\}^n_{\perp}$. δ is represented by a couple (Cns, \vec{f}) , where $Cns : \{0, 1\}^n \times \{0, 1\}^m \rightarrow \{0, 1\}$ is the characteristic function of the domain on which δ is defined, and $\vec{f} : \{0, 1\}^n \times \{0, 1\}^m \rightarrow \{0, 1\}^n$ gives the value of δ on this domain. Thus, $\delta = \lambda \vec{x}.(\mathbf{if} \ Cns(\vec{x}) = 1 \ \mathbf{then} \ \vec{f}(\vec{x}) \ \mathbf{else} \ \perp)$.
- $Init$ is the characteristic function of the set of initial states.

The 5-tuple $(n, m, \omega, \delta, Init)$ of a sequential machine \mathcal{M} can be obtained from its functional description, using a symbolic execution process such as the one detailed in [3]. The set of valid states of such a machine is the limit of the converging sequence of sets V_k defined by the equations:

$$\begin{aligned} V_0 &= \{s \in \{0, 1\}^n / Init(s) = 1\}, \text{ and} & (1) \\ V_{k+1} &= V_k \cup \{s' \in \{0, 1\}^n / \exists s \in V_k, \exists x \in \{0, 1\}^m, s' = \delta(s, x)\} \end{aligned} \quad (2)$$

The sequence (V_k) is computed using basic set operations, in addition with the *image computation*. We say that $Img(f, A) = \{f(x) / x \in A\}$ is the image of the function f on the set A . Equation 2 becomes $V_{k+1} = V_k \cup Img(\delta, V_k \times \{0, 1\}^m)$. Now we substitute each set by its characteristic function. Since δ is represented by a couple of functions (Cns, \vec{f}) such that $\delta = \lambda \vec{x}.(\mathbf{if} \ Cns(\vec{x}) = 1 \ \mathbf{then} \ \vec{f}(\vec{x}) \ \mathbf{else} \ \perp)$, the characteristic functions V_k are defined by:

$$V_0 = \text{Init}, \text{ and} \quad (3)$$

$$V_{k+1} = \lambda \vec{y}. (V_k(\vec{y}) \vee \text{Img}(\vec{f}, \lambda \vec{y}. \lambda \vec{x}. (V_k(\vec{y}) \wedge \text{Cns}(\vec{y}, \vec{x}))) (\vec{y})) \quad (4)$$

Figure 1 shows an algorithm that computes the fixed point defined by 3 and 4, where Boolean functions are represented by BDD's or TDG's. `Valid` denotes the set of reached states V_k after k iterations, `Range` is the result of the image computation, and `New` denotes the set of newly discovered states at each iteration. Since the usual Boolean operations on graphs have a polynomial complexity, the total cost of this algorithm finally depends on the cost of the operation “*Img*”. This is the object of the next section.

```
function compute-valid-states-1( $n, m, \omega, (\text{Cns}, \vec{f}), \text{Init}$ ) : TDG;
var Valid, New, Range : TDG;
Valid = Init;
New = Init;
while New  $\neq 0$  {
    Range =  $\text{Img}(\vec{f}, \text{Valid} \wedge \text{Cns})$ ;
    New = Range  $\wedge \neg \text{Valid}$ ;
    Valid = Valid  $\vee$  New;
}
return Valid;
```

Figure 1. Valid State Computation.

3 The Image Computation Problem

This section first gives basic results about the complexity of the image computation. Then it presents the notion of *range restrictor* and proposes a parametric algorithm that performs the image computation.

3.1 Complexity of the Image Computation

We call *range computation* the evaluation of the term $\text{Img}(\vec{f}, 1)$, which is an image computation on the whole domain. Image computation and range computation are closely related by the equality:

$$\text{Img}([f_1 \dots f_n], \chi) = \text{Img}([f_1 \dots f_n \chi], 1)[y_{n+1} \leftarrow 1], \quad (5)$$

so range and image computation have the same complexity. It can be studied using the composition problem, which consists in computing the graph of $\lambda \vec{x}.g(f_1(\vec{x}), \dots, f_n(\vec{x}))$ from the graphs of the Boolean functions f_1, \dots, f_n , and g . The following results show that image computation is the most difficult operation that appears in the algorithm presented in Section 2.2.

Theorem 3.1 *The composition problem is linearly reducible to the range computation problem.*

Proof. Let there be $\vec{x} = [x_1 \dots x_m]$ and $\vec{y} = [y_1 \dots y_n]$. From the graphs of the functions $\lambda \vec{x}.f_1(\vec{x}), \dots, \lambda \vec{x}.f_n(\vec{x}), \lambda \vec{y}.g(\vec{y})$, we define:

$$\vec{f} = \lambda(\vec{x} @ \vec{y}).[x_1 \dots x_m (y_1 \Leftrightarrow f_1(\vec{x})) \dots (y_n \Leftrightarrow f_n(\vec{x})) g(\vec{y})].$$

The graph of \vec{f} is built in $O(m + |g| + \sum_{k=1}^{k=n} |f_k|)$. The characteristic function χ of $\text{Img}(\vec{f}, 1)$ is:

$$\begin{aligned} \chi = \lambda[z_1 \dots z_{m+n+1}]. & (\exists \vec{x} \exists \vec{y} \left(\bigwedge_{k=1}^{k=m} (z_k \Leftrightarrow x_k) \right) \wedge \\ & \left(\bigwedge_{k=1}^{k=n} (z_{m+k} \Leftrightarrow (y_k \Leftrightarrow f_k(\vec{x}))) \right) \wedge \\ & (z_{m+n+1} \Leftrightarrow g(\vec{y}))). \end{aligned}$$

Then we have the identities:

$$\begin{aligned} \chi(z_1, \dots, z_m, 1, \dots, 1) &= (\exists \vec{x} \exists \vec{y} \left(\bigwedge_{k=1}^{k=m} (z_k \Leftrightarrow x_k) \right) \wedge \\ & \left(\bigwedge_{k=1}^{k=n} (y_k \Leftrightarrow f_k(\vec{x})) \right) \wedge g(\vec{y})) \\ &= (\exists \vec{y} \left(\bigwedge_{k=1}^{k=n} (y_k \Leftrightarrow f_k(z_1, \dots, z_m)) \right) \wedge g(\vec{y})) \\ &= g(f_1(z_1, \dots, z_m), \dots, f_n(z_1, \dots, z_m)) \end{aligned}$$

We obtain $g(f_1, \dots, f_n) = \lambda \vec{x}. \chi(x_1, \dots, x_m, 1, \dots, 1)$. This graph is built in $O(n)$ if the variable ordering satisfies $\{z_{m+1}, \dots, z_{m+n}, z_{m+n+1}\} < \{z_1, \dots, z_m\}$. \square

Since the composition problem is NP-hard, the image computation is NP-hard. Assuming that the variable ordering cannot be modified, we obtain the more precise following result.

Theorem 3.2 *Composition is a no polynomial problem if the ordering of the variables is fixed. More precisely: there exist some graphs f_1, \dots, f_n and g , using $O(n)$ variables, with $|g| + \sum_{k=1}^{k=n} |f_k| = O(n)$, and such that $|g(f_1, \dots, f_n)| = O(2^n)$.*

Proof. We consider the function $g = \lambda \vec{x}. (\bigwedge_{k=1}^{k=n} x_k)$ and the n functions $f_k = \lambda[y_1 \dots y_{2n}]. (y_k \Leftrightarrow y_{2n-k+1})$ on the $2n$ variables $y_1 < \dots < y_{2n}$. The graph of g is a n -vine of size n , and the graph of each function f_k has a constant size, so $|g| + \sum_{k=1}^{k=n} |f_k| = O(n)$. The composition $g(f_1, \dots, f_n)$ is the function $\lambda \vec{y}. (\bigwedge_{k=1}^{k=n} f_k(\vec{y}))$, that is $\lambda[y_1 \dots y_{2n}]. (\bigwedge_{k=1}^{k=n} (y_k \Leftrightarrow y_{2n-k+1}))$, and the graph of the term $((y_1 \Leftrightarrow y_{2n}) \wedge (y_2 \Leftrightarrow y_{2n-1}) \wedge \dots \wedge (y_n \Leftrightarrow y_{n+1}))$ with the variable ordering $y_1 < \dots < y_{2n}$ has a size in $O(2^n)$. \square

The image computation is at least exponential in the worst case. However, one can object that the graph of the term $((y_1 \Leftrightarrow y_{2n}) \wedge (y_2 \Leftrightarrow y_{2n-1}) \wedge \dots \wedge (y_n \Leftrightarrow y_{n+1}))$ is in $O(n)$ with the variable ordering $y_1 < y_{2n} < y_2 < y_{2n-1} < \dots < y_n < y_{n+1}$. One might think that the image computation can be performed more efficiently if we are able to find a variable ordering that minimizes the size of the resulting graph. This is not very simple: first, finding such a variable ordering is itself a NP-hard problem; second, there exist some functions of n variables whose graphs have sizes that are no polynomial with respect to n , whatever the variable ordering is [5]. Therefore the composition problem, and so the image computation, is not polynomial even if an oracle would provide dynamically a good ordering.

We have just shown that the image computation is not polynomial because of the memory needed to build the resulting graph. One might think that problems involving an image computation with a bounded output, for instance tests, have a lower complexity. Knowing whether $Img(\vec{f}, 1) = 1$ is one of these problems, and it is an interesting problem because it could be used to dramatically speed

up the image computation by recursion pruning [7, 9]. But Theorem 3.3 shows that this test is also NP-hard. Even testing whether an element of $\{0, 1\}^n$ belongs to $Img(\vec{f}, 1)$ is NP-complete.

Theorem 3.3 *Being given the graph of \vec{f} , to test whether $Img(\vec{f}, 1) = 1$ or not is NP-hard.*

Proof. Let $C = \bigwedge_{k=1}^{k=n} c_k$ be a 3-conjunctive normal form of n clauses c_k . Creating n variables x'_1, \dots, x'_n that do not occur in C and computing the n graphs of $(c_k \wedge x'_k)$ is a polynomial operation. Then $Img([c_1 \wedge x'_1 \dots c_n \wedge x'_n], 1) = 1$ iff $[1 \dots 1]$ belongs to $Img([c_1 \dots c_n], 1)$, that is iff C is satisfiable. \square

3.2 Image Computation Based on Range Restrictors

This section defines the range restrictors and shows how they can be used to perform the image computation. A parametric algorithm based on covers and on range restrictors is presented. Next part will discuss the choice of a range restrictor.

A *range restrictor* is an operator \mathbf{rr} that satisfies the equality $Img(\vec{f}, \chi) = Img(\vec{f} \mathbf{rr} \chi, 1)$ for any Boolean function $\chi \neq 0$ and for any Boolean vectorial function \vec{f} . Thus the computation of the term $Img(\vec{f}, \chi)$ comes down to computing $Img(\vec{f} \mathbf{rr} \chi, 1)$. This means the image computation problem becomes a range computation problem.

From this definition, a divide and conquer strategy can be used for computing the image. We say that a set of functions $(h_k)_{k \in K}$ is a *cover* if and only if $\lambda \vec{x}. (\bigvee_{k \in K} h_k(\vec{x})) = 1$. Then it is obvious that for any cover $(h_k)_{k \in K}$, we have $Img(\vec{f}, 1) = \bigcup_{k \in K} Img(\vec{f}, h_k)$. Figure 2 shows the functions “*Img*” and *range* that compute the image using a range restrictor and a cover.

```
function Img( $\vec{f}, c$ );
return range( $\vec{f} \mathbf{rr} c$ );

function range( $\vec{f}$ );
if  $\vec{f} = [\varepsilon_1(1) \dots \varepsilon_n(1)]$  then
  return  $\lambda \vec{y}. (\bigwedge_{k=1}^{k=n} \varepsilon_k(y_k))$ ;
let  $(h_k)_{k \in K}$  such that  $\lambda \vec{x}. (\bigvee_{k \in K} h_k(\vec{x})) = 1$  in
  return  $\lambda \vec{y}. (\bigvee_{k \in K} \mathbf{range}(\vec{f} \mathbf{rr} h_k)(\vec{y}))$ ;
```

Figure 2. Schema of the Function *Img*.

If at each step, the cover $(h_k)_{k \in K}$ contains at least two sets h_1 and h_2 such that $h_1 \not\subseteq h_2$ and $h_2 \not\subseteq h_1$, then the algorithm terminates. The complexity of this algorithm depends on the cover $(h_k)_{k \in K}$ and on the range restrictor **rr**. Assume that $\vec{f} = [f_1 \dots f_n]$ is a function from $\{0, 1\}^m$ into $\{0, 1\}^n$. If at each step the cover $(h_k)_{k \in K}$ is a partition of the domain, the number of recursions is bounded by the number of elements of the domain, that is 2^m . By using a cache that stores intermediate computations [7], this number can be dramatically reduced. This cache stores the couples (\vec{f}, c) such that c is the characteristic function of $\text{Img}(\vec{f}, 1)$. Instead of taking into account the whole vector \vec{f} in the cache, this technique can be refined by using the following equalities:

Theorem 3.4 *Let χ' be $\text{Img}([f_1 \dots f_{k-1} f_{k+1} \dots f_n], \chi)$. Then the function $\text{Img}([f_1 \dots f_{k-1} \varepsilon(1) f_{k+1} \dots f_n], \chi)$ is equal to:*
 $\lambda \vec{y}. (\varepsilon(y_k) \wedge \chi'(y_1, \dots, y_{k-1}, y_{k+1}, \dots, y_n)).$

Theorem 3.5 *Let χ' be $\text{Img}([f_1 \dots f_{k-1} f_{k+1} \dots f_n], \chi)$. Then the function $\text{Img}([f_1 \dots f_{k-1} \varepsilon(f_1) f_{k+1} \dots f_n], \chi)$ is equal to:*
 $\lambda \vec{y}. ((y_1 \Leftrightarrow \varepsilon(y_k)) \wedge \chi'(y_1, \dots, y_{k-1}, y_{k+1}, \dots, y_n)).$

Theorem 3.6 *Let $\chi'_1 = \text{Img}(\vec{f}_1, A_1)$ and $\chi'_2 = \text{Img}(\vec{f}_2, A_2)$. Then the function $\text{Img}(\lambda(\vec{x}_1 @ \vec{x}_2).(\vec{f}_1(\vec{x}_1) @ \vec{f}_2(\vec{x}_2)), A_1 \times A_2)$ is equal to:*
 $\lambda(\vec{y}_1 @ \vec{y}_2).(\chi_1(\vec{y}_1) \wedge \chi_2(\vec{y}_2)).$

Theorem 3.4 states that the constant components of \vec{f} can be eliminated, and Theorem 3.5 states that if a component or/and its negation occurs several times in \vec{f} , only one of these components has to be saved. These two elimination processes reduce the number of entries in the cache, and increase the hit ratio. So the number of recursive calls to the function **range** is reduced. Theorem 3.6 uses a partitioning of the vector into sets of functions of disjoint supports [9]. If the graphs f_1, \dots, f_n can be partitioned into a partition $\mathcal{P} = (\vec{f}_1, \dots, \vec{f}_q)$, where the vectors \vec{f}_k have disjoint supports of variables, then computing $\text{Img}([f_1 \dots f_n], 1)$ comes down to performing the product of the q characteristic functions of $\text{Img}(\vec{f}_k, 1)$. And in this manner the number of possible recursions is reduced from 2^m to $\sum_{k=1}^{k=q} 2^{s_k}$, where s_k is the size of the support of \vec{f}_k .

Let **eliminate-and-partition** be the function that applies to a vector $\vec{f} = [f_1 \dots f_n]$ and that returns a couple (c, \mathcal{P}) , such that: c is the characteristic function relative to the components that have

been eliminated from \vec{f} ; by convention, if no component of \vec{f} can be eliminated, then $c = 1$; \mathcal{P} is the partition of vectors of disjoint supports (these vectors are made of the remaining components of \vec{f}). The elimination process based on Theorem 3.4 and 3.5 can be done in $O(n \log n)$ with the TDG's, but it is more costly with the BDD's (see below the discussion on the *extended match test*). The partitioning of the remaining components can be done in $O((n + |F|)m \log m)$. Figure 3 shows the new function `range` using a cache.

```

function range( $\vec{f}$ );
var  $c'$ ;
let ( $c, \mathcal{P}$ ) = eliminate-and-partition( $\vec{f}$ ) in {
  case  $\mathcal{P}$  of {
    () :  $c' = 1$ ;
    ( $\vec{f}_1, \dots, \vec{f}_q$ ) :  $c' = \lambda \vec{y}. (\bigwedge_{k=1}^{k=q} \text{range-cache}(\vec{f}_k)(\vec{y}))$ ;
  }
  return  $\lambda \vec{y}. (c(\vec{y}) \wedge c'(\vec{y}))$ ;
}

function range-cache( $\vec{f}$ );
if  $\vec{f} = []$  or  $\vec{f} = [f]$  then return 1;
if is-in-cache?( $\vec{f}$ ) then return get-in-cache( $\vec{f}$ );
let  $c = \text{range-recurse}(\vec{f})$  in {
  put-in-cache( $\vec{f}, c$ );
  return  $c$ ;
}

function range-recurse( $\vec{f}$ );
let ( $h_k$ ) $_{k \in K}$  such that  $\lambda \vec{x}. (\bigvee_{k \in K} h_k(\vec{x})) = 1$  in
  return  $\lambda \vec{y}. (\bigvee_{k \in K} \text{range}(\vec{f} \text{ rr } h_k)(\vec{y}))$ ;

```

Figure 3. The Function `range` Using Elimination and Partitioning.

But instead of using an exact matching as in [7], we use an *extended matching* [10] based on the two following theorems:

Theorem 3.7 *Let χ be $\text{Img}([f_1 \dots f_k], A)$, and let the ε_k 's be identity or negation functions. We have the identity:*

$$\text{Img}([\varepsilon_1 \circ f_1 \dots \varepsilon_k \circ f_k], A) = \lambda [y_1 \dots y_k]. \chi(\varepsilon_1(y_1), \dots, \varepsilon_k(y_k))$$

Theorem 3.8 *Let χ be $\text{Img}([f_1 \dots f_k], A)$, and let σ be a permutation of the first k integers. We have the identity:*

$$\text{Img}([f_{\sigma(1)} \dots f_{\sigma(k)}], A) = \lambda[y_1 \dots y_k] \cdot \chi(y_{\sigma(1)}, \dots, y_{\sigma(k)})$$

These two properties mean that a couple (\vec{f}, c) in the cache denotes the range of $(k! \times 2^k)$ vectors of length k if the vector \vec{f} has k components. The complexity of the extended matching test depends directly on the Boolean function representation. With BDD's the problem is that testing whether a function f is the negation of g is in $O(\max(|f|, |g|))$. This implies that for any total ordering " \preceq " on BDD's, if the functions $\lambda f \cdot \lambda g \cdot ((f \preceq g) \wedge (g \preceq f))$ and $\lambda f \cdot \lambda g \cdot ((f = g) \vee (f = \neg g))$ are equal, then the Boolean function $\lambda f \cdot \lambda g \cdot (f \preceq g)$ can be computed at best in $O(\max(|f|, |g|))$. Then the extended match test between two BDD's vectors $\vec{f} = [f_1 \dots f_k]$ and $\vec{g} = [g_1 \dots g_k]$ can be done at best in $O((|F| + |G|)k \log k)$.

This is entirely different with TDG's. The graph of $(\neg f)$ is directly obtained from the one of f by inverting the type of the pointer to f , so testing whether a function f is the negation of g is in $O(1)$. We define a total ordering on the components of a vector $[f_1 \dots f_k]$ by: $f \preceq g$ if and only if $\text{add}(f) \leq \text{add}(g)$, where $\text{add}(f)$ is the address of the graph f if f is positive, and $\text{add}(f)$ is the address of the graph of $(\neg f)$ if f is negative. This ordering is done in $O(k \log k)$. When a vector of k components is rewritten into this form, one can decide in $O(k)$ whether the properties expressed by Theorem 3.7 and Theorem 3.8 can be used. Thus the extended match test between two TDG's vectors of k components is done in $O(k \log k)$. In fact, the ordering of the components can be directly performed by the function **eliminate-and-partition**. It will return a couple (c, \mathcal{P}) such that there is no repetition of a function and/or its negation in \mathcal{P} , and each vector of \mathcal{P} is ordered by \preceq .

3.3 Strict Range Restrictors

In this section we discuss the choice of a range restrictor. Since graph manipulations have a cost directly linked to the sizes of the graphs, the best range restrictor **rr** is the one that minimizes the size of the graph $(\vec{f} \text{ rr } \chi)$. Unfortunately the following theorem holds.

Theorem 3.9 *Being given Shannon's canonical form (or the graph) of a vector \vec{f} , the problem of finding a vector \vec{f}' satisfying $\text{Img}(\vec{f}', 1) =$*

$Img(\vec{f}, 1)$ and that has a minimal Shannon's canonical form (or minimal graph) is NP-hard.

Proof. We give here only the proof on Shannon's canonical form. Any vector \vec{f}' of n components such that $Img(\vec{f}', 1) = 1$ needs n independent variables, so $|\vec{f}'| \geq n$. The vectorial function identity Id has a size equal to n , and $Img(Id, 1) = 1$. Thus any vector \vec{f}' of n components that satisfies $Img(\vec{f}', 1) = 1$ and that has a minimal Shannon's canonical form has a size equal to n . It is then easy to show that any vector \vec{f}' of n components that satisfies $Img(\vec{f}', 1) = 1$ and has a minimal Shannon's canonical form is equal to one of the $n! \times 2^n$ functions $\lambda \vec{x}. [\varepsilon_1(x_{\sigma(1)}) \dots \varepsilon_n(x_{\sigma(n)})]$, where σ is a permutation of the first n integers. Therefore testing whether $Img(\vec{f}, 1) = 1$ can be done in two steps. First step computes the minimal vector \vec{f}' such that $Img(\vec{f}', 1) = Img(\vec{f}, 1)$. Second step tests whether \vec{f}' has the form described above, which can be done in $O(n^2)$. But a proof similar to the one of Theorem 3.3 shows easily that testing whether $Img(\vec{f}, 1) = 1$, where Shannon's canonical form of \vec{f} is given, is NP-hard. \square

A range restrictor that minimizes the size of $(\vec{f} \mathbf{rr} \chi)$ is tremendously costly. The problem is then to define a range restrictor that can be polynomially evaluated. A range restrictor that is polynomial on the recursive structure of Boolean functions (grammatical tree, Shannon's canonical form, BDD or TDG) has to be apply to itself recursively. Since a vectorial function is a tree (or a graph) made of elementary Boolean operations on vectorial functions, our problem can be expressed by:

- (1) Determine a range restrictor \mathbf{rr} such that, for all functions \vec{f} and \vec{g} , for all Boolean function χ , we have: $((\neg \vec{f}) \mathbf{rr} \chi) = \neg (\vec{f} \mathbf{rr} \chi)$, and $((\vec{f} \vee \vec{g}) \mathbf{rr} \chi) = ((\vec{f} \mathbf{rr} \chi) \vee (\vec{g} \mathbf{rr} \chi))$.

This is also equivalent to the following problem:

- (2) Determine a range restrictor \mathbf{rr} such that, for all functions \vec{f} and \vec{g} , for all Boolean function χ , we have: $((\vec{f} \circ \vec{g}) \mathbf{rr} \chi) = (\vec{f} \circ (\vec{g} \mathbf{rr} \chi))$.

Since any function \vec{f} from $\{0, 1\}^m$ into $\{0, 1\}^n$ is equal to $(\vec{f} \circ Id)$, where Id is the identity function, the range restrictors \mathbf{rr} satisfying

(2) are such that $(\vec{f} \mathbf{rr} \chi) = (\vec{f} \circ (Id \mathbf{rr} \chi))$ for any functions \vec{f} and χ . Therefore the range restrictors satisfying (2) can be written $\lambda \vec{f} . \lambda \chi . \lambda \vec{x} . \vec{f}(Fun(\chi, \vec{x}))$, where the function $Fun = \lambda \chi . \lambda \vec{x} . (Id \mathbf{rr} \chi)$ is a function from $\{0, 1\}^m$ into $\{0, 1\}^m$ that does not depend on \vec{f} . From the definition of a range restrictor, Fun is such that for all Boolean function $\chi \neq 0$, we have $Img(\lambda \vec{x} . Fun(\chi, \vec{x}), 1) = \chi$.

Let us define the *projectors*. A projector on a non empty set A is a function that maps the whole domain onto A . In other words, it takes a Boolean function χ and an element of $\{0, 1\}^m$ as input, and returns an element of $\{0, 1\}^m$; a projector P is such that for any Boolean function $\chi \neq 0$, $Img(\lambda \vec{x} . P(\chi, \vec{x}), 1) = \chi$. If P is a projector, then $\lambda \vec{f} . \lambda \chi . \lambda \vec{x} . \vec{f}(P(\chi, \vec{x}))$ is a range restrictor that satisfy (2). So defining a range restrictor that can be recursively evaluated consists in choosing a projector.

We then distinguish the *strict projectors*. A strict projector P is a projector such that, for any Boolean function χ , $\chi(\vec{x}) = 1$ implies that $P(\chi, \vec{x}) = \vec{x}$. In other words, the function $\lambda \vec{x} . P(\chi, \vec{x})$ on the set A is the identity. We say that \mathbf{srr} is a *strict range restrictor* if it can be written $\lambda \vec{f} . \lambda \chi . \lambda \vec{x} . \vec{f}(P(\chi, \vec{x}))$, where P is a strict projector. A strict range restrictor \mathbf{srr} is a range restrictor such that, for any Boolean function χ , $\chi(\vec{x}) = 1$ implies that $\vec{f}(\vec{x}) = (\vec{f} \mathbf{srr} \chi)(\vec{x})$. The behaviors of the functions \vec{f} and $(\vec{f} \mathbf{srr} \chi)$ on the set χ are identical. In particular, $(\vec{f} \mathbf{srr} 1) = \vec{f}$.

The behavior of a strict range restrictor is then characterized by its projector. For instance, we can take the function $\lambda \vec{x} . \lambda \chi . ((\chi(\vec{x}) \wedge \vec{x}) \vee (\neg \chi(\vec{x}) \wedge \vec{x}_0))$ as projector, where \vec{x}_0 is an element of $\{0, 1\}^m$ that satisfies χ [9]. The strict range restrictor \mathbf{srr} based on this projector is:

$$\mathbf{srr} = \lambda \vec{f} . \lambda \chi . \lambda \vec{x} . ((\chi(\vec{x}) \wedge \vec{f}(\vec{x})) \vee (\neg \chi(\vec{x}) \wedge \vec{f}(\vec{x}_0))) \quad (6)$$

Being given a function $\vec{f} = [f_1 \dots f_n]$, computing the graph of $(\vec{f} \mathbf{srr} \chi)$ consists in: finding an \vec{x}_0 of the domain $\{0, 1\}^m$ such that $\chi(\vec{x}_0) = 1$, which is done in $O(m)$; computing the element $[v_1 \dots v_n]$ of the co-domain $\{0, 1\}^n$ that is equal to $\vec{f}(\vec{x}_0)$, which is done in $O(n \times m)$; then building the n graphs $(\chi(\vec{x}) \wedge f_k(\vec{x})) \vee (\neg \chi(\vec{x}) \wedge v_k)$, which is done in $O(|\chi| \times \sum_{k=1}^{k=n} |f_k|)$. Thus computing the graph of $(\vec{f} \mathbf{srr} \chi)$ is in $O(|\chi| \times \sum_{k=1}^{k=n} |f_k|)$, which is quite satisfying. But we want to have a strict range restrictor \mathbf{srr} such that the size of $(\vec{f} \mathbf{srr} \chi)$ is kept relatively low with regard to the sizes of \vec{f} and χ .

A strict projector that minimizes the size of $(\vec{f} \circ \lambda \vec{x}.P(\vec{x}, \chi))$ is certainly not polynomial. The heuristic we use is minimizing the size of Shannon's canonical form of $(\vec{f} \circ \lambda \vec{x}.P(\vec{x}, \chi))$. Of course, the only relation between the size of a graph and the size of its Shannon's canonical form is an inequality. This means minimizing Shannon's canonical form of $(\vec{f} \circ \lambda \vec{x}.P(\vec{x}, \chi))$ does not give a solution that minimizes the graph of $(\vec{f} \circ \lambda \vec{x}.P(\vec{x}, \chi))$, but it can be a good practical approximation. For this purpose, the next section presents a strict range restrictor called "constrain". It is generated by a well adapted strict projector: we will see that "constrain" is optimal on Shannon's canonical form, and that it has a quadratic complexity.

3.4 The Constrain Operator

In order to present the strict range restrictor called "constrain", we first define the strict projector CI (for *Closest Interpretation*). There is a logical definition of the function CI , based on the resolution of a Boolean equation, but we give here a more intuitive topological definition.

3.4.1 Topological Definition of the Closest Interpretation

We consider the space of the interpretations $\{0, 1\}^n$ (n is possibly equal to \aleph , except when specified) as the domain D . We define, for any two elements $\vec{x} = [v_1, v_2, \dots]$ and $\vec{y} = [w_1, w_2, \dots]$ of D , the function $d(\vec{x}, \vec{y}) = \sum_{k=1}^{k=n} |v_k - w_k|/k2^k$.

Theorem 3.10 *The function d is an ultra-metric distance, that is a distance such that for any elements x, y and z , $d(x, z) = d(y, z)$ if and only if $x = y$.*

Proof. It is obvious that d is a distance: the function d is positive and symmetric ; $d(\vec{x}, \vec{y}) = 0$ if and only if $v_k = w_k$ for all k , which is true if and only if $\vec{x} = \vec{y}$; if $\vec{z} = [u_1, u_2, \dots]$ is any third element of D , since $|v_k - w_k| = |(v_k - u_k) - (w_k - u_k)| \leq |v_k - u_k| + |w_k - u_k|$, we obtain easily $d(\vec{x}, \vec{y}) \leq d(\vec{x}, \vec{z}) + d(\vec{z}, \vec{y})$. Now, assume that $\vec{x} \neq \vec{y}$, and let h be the minimal integer k such that $v_k \neq w_k$. We have:

$$|d(\vec{x}, \vec{z}) - d(\vec{y}, \vec{z})| = \left| \sum_{k=1}^{k=n} (|v_k - u_k| - |w_k - u_k|)/k2^k \right|$$

$$\begin{aligned}
&\geq 1/h2^h - \left| \sum_{k=h+1}^{k=n} (|v_k - u_k| - |w_k - u_k|)/k2^k \right| \\
&\geq 1/h2^h - \sum_{k=h+1}^{k=n} \left| |v_k - u_k| - |w_k - u_k| \right|/k2^k \\
&\geq 1/h2^h - \sum_{k=h+1}^{k=n} 1/k2^k \\
&> 1/h2^h - \left(\sum_{k=h+1}^{k=\infty} 1/2^k \right)/h + 1 \\
&= (1/h - 1/(h+1))/2^h \\
&> 0
\end{aligned}$$

So $\vec{x} \neq \vec{y}$ implies that $d(\vec{x}, \vec{z}) \neq d(\vec{y}, \vec{z})$. Since d is a distance, the converse is obvious. \square

Theorem 3.11 *Let \vec{x} be an element of D . The relation noted $\preceq_{\vec{x}}$, and defined for any elements \vec{y}_1 and \vec{y}_2 of D as $\vec{y}_1 \preceq_{\vec{x}} \vec{y}_2$ if and only if $d(\vec{x}, \vec{y}_1) \preceq_{\vec{x}} d(\vec{x}, \vec{y}_2)$, is a total ordering on D .*

Proof. Since d is a bounded distance, $d(\vec{x}, \vec{y}_1)$ and $d(\vec{x}, \vec{y}_2)$ are finite for any elements \vec{y}_1 and \vec{y}_2 of D , so we have $\vec{y}_1 \preceq_{\vec{x}} \vec{y}_2$ or $\vec{y}_2 \preceq_{\vec{x}} \vec{y}_1$. The relation $\preceq_{\vec{x}}$ is transitive because d is a distance. If $\vec{y}_1 \preceq_{\vec{x}} \vec{y}_2$ and $\vec{y}_2 \preceq_{\vec{x}} \vec{y}_1$, then we have $d(\vec{x}, \vec{y}_1) = d(\vec{x}, \vec{y}_2)$, which implies that $\vec{y}_1 = \vec{y}_2$ because d is an ultra-metric distance. \square

Theorem 3.12 *We consider here that the domain D is finite. Let \vec{x} be an element of D and χ be a non null Boolean function on the domain D . Among the elements \vec{x}' of D such that $\chi(\vec{x}') = 1$, there exists a unique element that minimizes the distance d with \vec{x} .*

Proof. Let S be the subset of D whose χ is the characteristic function. This set is non empty because $\chi \neq 0$. The relation $\preceq_{\vec{x}}$ is a total ordering on S , and since the distance d is bounded, S has a most lower bound \vec{x}_{\min} . Since S is finite, \vec{x}_{\min} belongs to S , so $\chi(\vec{x}_{\min}) = 1$. \square

Theorem 3.12 allows us to define the function CI . For any non null Boolean function χ , $CI(\vec{x}, \chi)$ is the closest element —with respect to the distance d — of \vec{x} that satisfies χ . It is easy to verify that CI is a strict projector.

3.4.2 Definition and Evaluation of the Constrain Operator

We define the operator “constrain” , which is noted \downarrow , as:

$$\downarrow = \lambda \vec{f}. \lambda \chi. (\vec{f} \circ \lambda \vec{x}. CI(\vec{x}, \chi)). \quad (7)$$

Since CI is a strict projector, the operator \downarrow is a strict range restrictor. Figure 4 shows the algorithm that computes the graph (or Shannon’s canonical form) of $\lambda \vec{x}. f(CI(\vec{x}, c))$ from the graphs (or Shannon’s canonical forms) of the two Boolean functions f and c [9]. In this figure, “c.root” is the variable of minimal ordering that occurs in the graph (or Shannon’s canonical form) c . This algorithm exploits the properties of strict image restrictors given in Section 3.3. If a cache is used to avoid redundant computations, then the complexity is in $O(|f| \times |c|)$. Being given a vector $\vec{f} = [f_1 \dots f_n]$, the computation of $\vec{f} \downarrow c$ is done by building the vector $[f_1 \downarrow c \dots f_n \downarrow c]$, what is done in $O(|\vec{f}| \times |c|)$. The complexity of this strict range restrictor is better than the one obtain in the previous section.

```
function constrain(f, c);
if c = 0 then error;
return cnst(f, c);

function cnst(f, c);
if c = 1 or f = 0 or f = 1 then return f;
if f = c then return 1;
if f = ¬ c then return 0;
let x = c.root in {
  if c[x ← 0] = 0 then return cnst(f[x ← 1], c[x ← 1]);
  if c[x ← 1] = 0 then return cnst(f[x ← 0], c[x ← 0]);
  return (¬ x ∧ cnst(f[x ← 0], c[x ← 0])) ∨
         (x ∧ cnst(f[x ← 1], c[x ← 1]));
}
```

Figure 4. The Operator “Constrain”.

Moreover, we can show that the operator \downarrow is optimal on Shannon’s canonical form, in the sense presented by Theorem 3.13.

Theorem 3.13 *Among the functions \vec{f}' satisfying $(\forall \vec{x}, \chi(\vec{x}) \Rightarrow \vec{f}'(\vec{x}) = \vec{f}(\vec{x}))$ and $Img(\vec{f}, \chi) = Img(\vec{f}', 1)$, the function $(\vec{f} \downarrow \chi)$ has a minimal Shannon’s canonical form.*

4 Algorithms

Section 3.2 has shown an algorithm for the image computation, dependent from a range restrictor and a cover. This section discusses two instances of this algorithm, called partition of the co-domain, and partition of the domain. We also give several ideas about the covers that could be used according to structural properties of the graph \vec{f} .

4.1 Using a Partition of the Co-Domain.

To get a cover $(h_k)_{k \in K}$ of the domain, we can either work directly on the domain, or use a cover of the co-domain. Let $(g_k)_{k \in K}$ be a cover. The function $(g_k \circ \vec{f})$ is the characteristic function of the elements \vec{x} of the domain such that $\vec{f}(\vec{x})$ belongs to g_k . Thus $(g_k \circ \vec{f})_{k \in K}$ is a cover of the domain.

But the function composition is NP-hard, so we must carefully choose $(g_k)_{k \in K}$ in order to get a low composition cost. Let us choose [9] the partition $\{g_0, g_1\}$ such that $g_0 = \lambda \vec{y}.(\neg y_k)$ and $g_1 = \lambda \vec{y}.y_k$. This couple divides the co-domain into two subsets of equal size. We have $(g_0 \circ \vec{f}) = \neg f_k$, and $(g_1 \circ \vec{f}) = f_k$. If **srr** is a strict range restrictor, then we have $([f_1 \dots f_n] \text{ srr } \varepsilon(f_k))$ equal to $([f_1 \dots f_{k-1}] \text{ srr } \varepsilon(f_k)) @ [\varepsilon(1)] @ ([f_{k+1} \dots f_n] \text{ srr } \varepsilon(f_k))$. Figure 5 shows the function **range-recurse** using co-domain partitioning. With this function **range-recurse**, the number of recursive calls to **range** is bounded by the number of elements of $Img(\vec{f}, 1)$. So if at each step the machine can reach only a small number of states, this algorithm is very efficient.

```
function range-recurse( $\vec{f}$ );
let  $[f_1 \dots f_n] = \vec{f}$  and
     $k = \text{choose-index}(\vec{f})$  and
     $\vec{f}' = [f_1 \dots f_{k-1} f_{k+1} \dots f_n]$  in
return  $\lambda \vec{y}.((\neg y_k \wedge \text{range}(\vec{f}' \text{ srr } \neg f_k)(y_1, \dots, y_{k-1}, y_{k+1}, \dots, y_n))$ 
     $\vee (y_k \wedge \text{range}(\vec{f}' \text{ srr } f_k)(y_1, \dots, y_{k-1}, y_{k+1}, \dots, y_n))$ );
```

Figure 5. The Function **range-recurse** Using Co-domain Partitioning.

The behavior of this algorithm depends on the function **choose-index**. For instance, it may always return 1. A better solution is to use

heuristics to increase the occurrences of constants in $(\vec{f} \text{ srr } \varepsilon(f_k))$, or to keep the size of the graph $(\vec{f} \text{ srr } \varepsilon(f_k))$ as low as possible.

The function `choose-index` can return the index of the component that has the smallest number of variables, which is done in $O(|\vec{f}|m \log m)$. In this way, we avoid to mix the supports of the components of $(\vec{f} \text{ srr } \varepsilon(f_k))$ so that Theorem 3.6 can be applied more often. However, if all components share nearly the same support, this heuristic is irrelevant.

Another idea is to return the index k of the “most discriminating” component f_k . The *discrimination* of a function f of m variables can be measured by the number $Dis(f) = 1 - |2n_1 - 2^m|/2^m$, where n_1 is the number of interpretations of the m variables that evaluate the function f to 1. The discrimination of f is computed in $O(|f|)$. $Dis(f)$ is equal to 0 when f is a constant, and is equal to 1 when f separates the domain into two subsets of equal size. The heuristic consists in choosing the component that has the greatest discrimination power, which is done in $O(\sum_{k=1}^{k=n} |f_k|)$. The aim is to avoid deep recursions. However this technique does not take into account the size of graphs, so it can rise to a dangerous graph expansion.

Since $(\vec{f} \text{ srr } \varepsilon(f_k))$ has a size in $O(|\vec{f}'| \times |f_k|)$, the function `choose-index` can also return the index of the component that has the smallest graph, which is done in $O(\sum_{k=1}^{k=n} |f_k|)$. Most of the time experiences show that this last heuristic gives better results. Section 5 gives experimental results of this technique.

4.2 Using a Partition of the Domain

Let us choose [10] the partition $\{h_0, h_1\}$ such that $h_0 = \lambda\vec{x}.(\neg x_k)$ and $h_1 = \lambda\vec{x}.x_k$. This couple separates the domain into two subsets of equal size. Then we have $(\vec{f} \downarrow h_0) = \vec{f}[x_k \leftarrow 0]$ and $(\vec{f} \downarrow h_1) = \vec{f}[x_k \leftarrow 1]$, which are computed in $O(|\vec{f}|)$. Figure 6 shows the function `range-recurse` using this domain partition

```
function range-recurse( $\vec{f}$ );
let  $x$  be an atom occurring in  $\vec{f}$  in
  return  $\lambda\vec{y}.$ (range( $\vec{f}[x \leftarrow 0]$ )( $\vec{y}$ )  $\vee$  range( $\vec{f}[x \leftarrow 1]$ )( $\vec{y}$ ));
```

Figure 6. The Function `range-recurse` Using a Simple Domain Partitioning.

With this function `range-recurse` there are 2^m possible recursions, but the behavior of this algorithm essentially depends on the choice of the atom x . We presents two ideas for reducing the number of recursive call to `range`.

The first idea comes from the following remark: if we want to exploit at best the cache, we must avoid to create new vertices for building the vectors $\vec{f}[x \leftarrow 0]$ and $\vec{f}[x \leftarrow 1]$. This is done by choosing the atom x as the atom occurring in \vec{f} with the minimal rank. Then the choice of x and the computation of $\vec{f}[x \leftarrow 0]$ and $\vec{f}[x \leftarrow 1]$ is only in $O(k)$, where k is the number of components of \vec{f} . Choosing the atom x in this manner makes the algorithm be a depth-first parallel traversal of the graphs of f_1, \dots, f_n . This means no vertex is created for building the vectors during the recursions. The only vertices that are needed are the one's of the characteristic functions stored in the cache. Since the vector taken as input by the function `range-recurse` is built from sub-graphs of the initial vector $\vec{f} = [f_1 \dots f_n]$, the number of recursions is obviously bounded by $\prod_{k=1}^{k=n} |f_k|$, which is exponential. We actually think that thanks to the function `eliminate-and-partition` and the extended match test, the number of recursions is directly related to $|\vec{f}|$. So if the graphs of the initial vector \vec{f} are relatively small, or if there are many sharings between these graphs, then this algorithm is very efficient. Section 5 gives experimental results of this technique.

The second idea is motivated by the last remark above. Since choosing the atom that has the minimal rank gives an algorithm whose number of recursions is related to the size of \vec{f} , it is interesting to look for another ordering of the variables with which the size of \vec{f} is reduced. Computing an optimal ordering for which $|\vec{f}|$ is minimum is NP-hard, consequently, we have to use heuristics. The problem can now be stated: being given the graphs of a vectorial function \vec{f} , compute polynomially a new ordering of the variables that reduces the number of vertices required to denote the same function \vec{f} .

We propose now a polynomial algorithm for re-ordering the variables in order to reduce the size of \vec{f} . Moreover, this algorithm guarantees that the new ordering cannot be worse than the previous one. In other words, the number of vertices obtained with the new ordering is less than or equal to the one obtained with the previous ordering. Let x_1, \dots, x_m be the variables occurring in the graphs of \vec{f} , with the ordering $x_1 < \dots < x_m$. The algorithm is iterative: at the k -th step, the first $k - 1$ variables of the new ordering have been

found, and the k -th variable of the new ordering has to be chosen among the remaining $m - k + 1$ variables. After m iteration, the new ordering has been entirely computed, just as the corresponding re-ordered graphs. Figure 7 shows the function `re-order` that returns a new ordering and the corresponding graphs. In this figure, $\vec{f}[x_k \leftarrow x]$ is the vector $((\neg x \wedge \vec{f}[x_k \leftarrow 0]) \vee (x \wedge \vec{f}[x_k \leftarrow 1]))$. Evaluating `re-order`(1, (x_1, \dots, x_m) , f) insures us to get a better or equal ordering. The complexity of this algorithm is in $O(m^2 \times |\vec{f}|)$. It can also be iterated to improve the new ordering until a fix point is reached. We do not know whether it converges to an optimal ordering or not. Anyway this algorithm can reduce dramatically the size of $|\vec{f}|$ on practical examples.

```

function re-order( $k$ , order,  $\vec{f}$ );
if  $k = m$  then return(order,  $\vec{f}$ );
let  $(x_1, \dots, x_{k-1}, x_k, \dots, x_m) = \text{order}$  and
     $x$  be an atom such that  $x_{k-1} < x < x_k$  in {
         $j = \min_{k \leq j \leq m} |\vec{f}[x_j \leftarrow x]|$ ;
        return re-order( $k + 1$ ,
             $(x_1, \dots, x_{k-1}, x_j, x_k, \dots, x_{j-1}, x_{j+1}, \dots, x_m)$ ,
             $\vec{f}[x_j \leftarrow x]$ );
    }

```

Figure 7. To Re-order the Variables to Reduce a Set of Graphs.

4.3 Other Ideas on Covers

We shortly present two other covers. Both of them have not been implemented.

If several graphs among $[f_1 \dots f_n]$ share a sub-graph g , then it can be interesting to choose the cover $(g, \neg g)$: the sub-graph g is replaced with constants in the graphs of $(\vec{f} \text{ srr } g)$ and $(\vec{f} \text{ srr } \neg g)$, and we obtain two easier image computations. This heuristic is efficient when many components of \vec{f} share the same graph g , or when g is large enough to strongly constrain $(\vec{f} \text{ srr } g)$ and $(\vec{f} \text{ srr } \neg g)$. Detecting the graphs that are shared can be done as follows. A counter initialized to zero is associated with each vertex. Then a traversal of the graphs of \vec{f} is performed, in such a way that each time a vertex is encountered, its counter is incremented, and if a vertex has already been encountered then its successors are not traversed

anymore. At the end of this process the counter of a vertex indicates the number of pointers to this vertex. A second traversal of the graphs of \vec{f} selects the most often shared graphs. Then, their sizes are computed, and a weighted function decides which graph must be chosen to simplify the image computation. This decision is made in $O(|\vec{f}|)$.

Another idea is to choose a graph g such that $(\vec{f} \text{ srr } g)$ and $(\vec{f} \text{ srr } \neg g)$ are vectors that can be partitioned into several sub-vectors of disjoint support of variables. However, the decision function for this heuristic seems much more complicated.

5 Experimental Results and Discussion

5.1 Final Algorithm

Figure 8 gives the algorithm for valid state computation using characteristic functions and the function `range`. At each step, `Current` contains all the new reached states (the set `New`), in addition to some already discovered states (belonging to `Valid`). Therefore the algorithm terminates and computes the valid states. Since the size of the vector $(\vec{f} \downarrow (\text{Current} \wedge \text{Cns}))$ is in $O(|\vec{f}| \times |\text{Current}| \times |\text{Cns}|)$, the idea is to choose a graph `Current` as small as possible [8, 9]. Unfortunately, computing a minimal graph f satisfying $(\text{New} \Rightarrow f \Rightarrow \text{Valid})$ is a NP-hard problem. Thus, we use a non optimal function [8] (which can be expressed in term of the “restrict” operator [8, 10]) whose complexity is in $O(|\text{New}| \times |\text{Valid}|)$.

```
function compute-valid-states-2( $n, m, \omega, (Cns, \vec{f}), Init$ ) : TDG;
var Valid, New, Current, Range : TDG;
Valid = Init;
New = Init;
while (New  $\wedge$  Cns)  $\neq$  0 do {
    choose Current such that (New  $\Rightarrow$  Current  $\Rightarrow$  Valid);
    Range = range( $\vec{f} \downarrow$  (Current  $\wedge$  Cns));
    New = Range  $\wedge$   $\neg$  Valid;
    Valid = Valid  $\vee$  New;
}
return Valid;
```

Figure 8. Symbolic Computation of the Valid States.

To reduce the time required by the function `range`, the following idea can be used [8]: since the only interesting states are the one's that do not yet belong to `Valid`, we can compute the domain D with which new states only can be reached. The characteristic function of this domain is the function $\lambda(s@p).(\mathbf{Current}(s) \wedge \mathit{Cns}(s, p) \wedge \neg \mathbf{Valid}(\vec{f}(s, p)))$. The set of new states `New` is directly equal to $\mathit{range}(F \downarrow D)$. In some cases, this method drastically reduces the number of recursive calls to the function `range`. In particular, an algorithm using this idea can be stopped when $D = 0$, what avoids to perform the last image computation that detects the fix point. But computing D requires the composition $(\mathbf{Valid} \circ \vec{f})$, which can be very costly. So we no longer use this operation, contrary to [7].

5.2 Experimental Results

We compare here the two algorithms proposed in Section 4.1 and Section 4.2. Both algorithms are based on the strict range restrictor “constrain”. The first algorithm uses a co-domain partition, where the component chosen for constraining the vector is the smallest one. The second algorithm uses a domain partition, where the atom that is chosen for splitting the domain is the one that has the minimal rank. Both algorithms are written in C, and the performances are given for a SPARC Station.

\mathcal{M}	features			valid states computation		
	# regs	# ins	# valids	depth	time-codp	time-dp
<i>s298</i>	14	3	218	19	0.3	0.4
<i>s344</i>	15	9	2625	7	4.2	2.8
<i>s349</i>	15	9	2625	7	4.3	2.8
<i>s382</i>	21	3	8865	151	7.4	6.2
<i>s400</i>	21	3	8865	151	7.3	6.2
<i>s420</i>	16	19	17	17	0.2	0.3
<i>s444</i>	21	3	8865	151	7.2	6.1
<i>s526</i>	21	3	8868	151	7.1	6
<i>s641</i>	19	35	1544	7	11.5	5.6
<i>s713</i>	19	35	1544	7	11.2	5.8
<i>s838</i>	32	35	17	17	0.4	0.5
<i>cbp16</i>	16	17	$6.5 \cdot 10^4$	2	0.4	0.4
<i>cbp32</i>	32	33	$4.3 \cdot 10^9$	2	1.7	1.3
<i>key</i>	56	62	$7.2 \cdot 10^{16}$	2	2.6	1.2
<i>stage</i>	64	113	$1.8 \cdot 10^{19}$	2	<i>memfull</i>	84.3
<i>sbc</i>	28	40	$1.5 \cdot 10^5$	10	1841.1	307.6
<i>clm1</i>	33	14	$3.8 \cdot 10^5$	397	76.5	184.7
<i>clm2</i>	33	388	$1.6 \cdot 10^5$	412	85.4	149.6
<i>clm3</i>	32	382	$3.3 \cdot 10^6$	279	500.2	<i>memfull</i>
<i>mm10</i>	30	13	$1.8 \cdot 10^8$	4	<i>memfull</i>	3.4
<i>mm20</i>	60	23	$1.9 \cdot 10^{17}$	4	<i>memfull</i>	12
<i>mm30</i>	90	33	$2 \cdot 10^{26}$	4	<i>memfull</i>	30.2

Figure 9. Experimental Results.

Figure 9 gives the CPU time in seconds needed to compute the set of valid states of some digital circuits (*mem full* indicates that more than 20 Mb had been allocated). For all circuits, **# ins** is the number of inputs, **# regs** the number of state variables, **depth** is the number of iterations, **# valid** is the number of valid states, **time-codp** and **time-dp** are the CPU times for the algorithm based on co-domain partitioning and for the algorithm based on domain partitioning respectively.

Figure 10 gives for both methods (codomain splitting and domain splitting) the number of recursions (**# rec**), the number of hits in the cache (**# match**) using extended matching, and the number of hits that use Theorem 3.7 or/and 3.8 (**# ext**). This last data enables us to evaluate the gain brought by extended matching: more than

20% of the hits are due to extended matching.

\mathcal{M}	codomaine split			domaine split		
	# rec	# match	# ext	# rec	# match	# ext
<i>s298</i>	32	24	0	32	27	11
<i>s344</i>	814	190	118	780	431	261
<i>s349</i>	814	190	118	780	431	261
<i>s382</i>	674	403	136	376	363	151
<i>s400</i>	674	403	136	376	363	151
<i>s420</i>	23	8	0	35	7	0
<i>s444</i>	672	399	161	376	363	185
<i>s526</i>	840	457	188	709	563	146
<i>s641</i>	1109	237	1	2184	1716	748
<i>s713</i>	1103	239	2	2349	1832	543
<i>s838</i>	23	8	0	35	7	0
<i>cbp16</i>	15	0	0	16	0	0
<i>cbp32</i>	61	40	0	32	13	0
<i>key</i>	109	106	0	2	1	0
<i>stage</i>	—	—	—	7339	10520	1391
<i>sbc</i>	122310	33343	4815	63762	63829	17376
<i>clm1</i>	2153	1764	344	19390	20361	1355
<i>clm2</i>	2363	2047	386	15972	17863	1381
<i>clm3</i>	29776	21981	5754	—	—	—
<i>mm10</i>	—	—	—	494	419	45
<i>mm20</i>	—	—	—	1094	969	105
<i>mm30</i>	—	—	—	1694	1519	165

Figure 10. Analysis of the Extended Match Test.

There are circuits that can be treated by only one of the algorithms. The circuit *clm3* can be treated only with co-domain partitioning: at each step during the computation, only a few states are reached. The *MinMax* [7, 9, 16] circuits (*mm10*, *mm20*, *mm30*) can be treated only with domain partitioning: the hit ratio in the cache is very high (see Figure 10) for this algorithm, which is not the case for the co-domain partitioning, and the number of states that are reached at each step is very large.

5.3 Other Approaches

The first image computation algorithm we have introduced [8] came directly from the definition of the image. Let $\vec{f} = [f_1 \dots f_n]$ be a vectorial Boolean function from $\{0, 1\}^m$ into $\{0, 1\}^n$, and χ' be the characteristic function of $Img(F, \chi)$. The function χ' is defined by:

$$\chi' = \lambda \vec{y}. (\exists \vec{x} \chi(\vec{x}) \wedge \left(\bigwedge_{k=1}^{k=n} y_k \Leftrightarrow f_k(\vec{x}) \right)) \quad (8)$$

From this identity, the graph χ' can be computed in two steps. The first step consists in building the graph of $(\chi(\vec{x}) \wedge (\bigwedge_{k=1}^{k=n} y_k \Leftrightarrow f_k(\vec{x})))$. The second step consists in eliminating the quantified atoms x_1, \dots, x_m associated with \vec{x} . The complexity of the first step is in $O(|\chi| \times 2^n \times \prod_{k=1}^{k=n} |f_k|)$, so exponential. Since eliminating a quantified atom from a graph g can give a graph of size $((|g| - 1)/2)^2$, we can prove that the complexity of the second step is also exponential. However, experience shows that the elimination of quantified atoms is feasible with practical examples. So, the cost of this computation depends mainly on the size of $(\chi(\vec{x}) \wedge (\bigwedge_{k=1}^{k=n} y_k \Leftrightarrow f_k(\vec{x})))$. If this graph can be built, then the computation of c is very efficient [6, 8, 16]. The problem is that the term $(\bigwedge_{k=1}^{k=n} y_k \Leftrightarrow f_k(\vec{x}))$ represents the transition relation of the machine, which cannot be built for complex machines. Experience shows that this method cannot be applied for non trivial machines [8, 16], or for machines whose the transition function's graphs are not regular enough [6].

In [16] Touati *et al.* propose heuristics that can reduce the cost of the image computation based on 8. The idea is to order the variables and the computations of the terms $(y_k \Leftrightarrow f_k(\vec{x}))$. Thus, quantified variables are eliminated as soon as possible, and the intermediate product can be kept relatively small. If all components f_k have nearly the same support of variables, the resulting ordering is irrelevant. However this technique is effective on many practical examples, which are sequential machines whose transition functions have weakly correlated components.

5.4 Conclusion

In this paper, we have shown that the symbolic computation of the valid states of a sequential machine depends on the image computation, which is the only no polynomial operation with respect to the

sizes of the graphs that denotes the sequential machine. We have proposed a parametric algorithm that performs the image computation. It is sufficient to choose a range restrictor and a cover to get an instance of this algorithm. Some classes of range restrictors have been studied, and we have justified the choice of the range restrictor “constrain”. Then several instances of the algorithm scheme have been proposed.

We have mainly compared two instances. They give good results, but there are some machines that can be efficiently treated by only one. Since both instances use the same skeleton and the same cache, both techniques can be mixed. The problem is to decide which one must be applied at each step of the recursion.

Our experiences seem to show that the main point to be studied for improving the valid state computation remains the variable ordering. Graph representation enables us to efficiently manipulate Boolean functions because of its canonicity. But the price of the canonicity is a fixed ordering of the variables, and the size of the graphs depends heavily of this ordering. Ordering the variables of combinatorial functions is a difficult problem, but ordering the variables for treating sequential machines is much more complicated. What can happen (and what has really happened with some machines) is that while the size of the graphs describing the sequential machine is low, the memory used to compute the valid states is prohibitive (for example the size of the characteristic function of the valid states increases exponentially during the computation). But with some other ordering, the size of the required memory is reasonable. This means that variable ordering should be *dynamically modified*, using a function similar to Section 4.2’s one, or using ordering heuristics presented in [16].

References

- [1] C. Berthet, O. Coudert, J. C. Madre, “New Ideas on Symbolic Manipulations of Finite State Machines”, in *Proc. of ICCD’90*, Sept. 1990.
- [2] J. P. Billon, “Perfect Normal Forms for Discrete Functions”, *BULL Report N° 87019*, March 1987.

- [3] J. P. Billon, J. C. Madre, “Original Concepts of PRIAM, an Industrial Tool for Efficient Formal Verification of Combinatorial Circuits”, in *The Fusion of Hardware Design and Verification*, G. J. Milne Editor, North Holland, 1988.
- [4] S. Bose, A. Fisher, “Automatic Verification of Synchronous Circuits Using Symbolic Logic Simulation and Temporal Logic”, in *Proc. of the IFIP Int’l Workshop, Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, November 1989.
- [5] R.E. Bryant, “Graph-based Algorithms for Boolean Functions Manipulation”, *IEEE Transactions on Computers*, Vol C35, 1986.
- [6] S. Burch, E. M. Clarke, K. L. McMillan, “Symbolic Model Checking: 10^{20} States and Beyond”, in *Proc. of LICS*, 1990.
- [7] H. Cho, G. Hachtel, S. W. Jeong, B. Plessier, E. Schwarz, F. Somenzi, “ATPG Aspect of FSM Verification”, in *Proc. of ICCAD’90*, Santa-Clara CA, November 1990.
- [8] O. Coudert, C. Berthet, J. C. Madre, “Verification of Synchronous Sequential Machines Based on Symbolic Execution”, in J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, Vol. 407 of *LNCS*, Springer-Verlag, 1989.
- [9] O. Coudert, C. Berthet, J. C. Madre, “Verification of Sequential Machines Using Boolean Functional Vectors”, in *Proc. of the IFIP Int’l Workshop, Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, November 1989.
- [10] O. Coudert, J. C. Madre, “A Unified Framework for the Formal Verification of Sequential Circuits”, in *Proc. of ICCAD’90*, Santa-Clara CA, November 1990.
- [11] M. Fujita, H. Fujisawa, N. Kawato, “Evaluations and Improvements of a Boolean Comparison Method Based on Binary Decision Diagrams”, in *Proc. of ICCAD’88*, Santa-Clara CA, November 1988.
- [12] S. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1978.

- [13] A. Ghosh, S. Devadas, A. R. Newton, "Test Generation for Highly Sequential Circuits", In *Proc. ICCAD'89*, Santa-Clara CA, Nov. 1989.
- [14] B. Lin, H. J. Touati, A. R. Newton, "Don't Care Minimization of Multi-Level Sequential Logic Networks", In *Proc. of ICCAD'90*, Santa-Clara CA, November 1990.
- [15] C. Pixley, "A Computational Theory and Implementation of Sequential Hardware Equivalence", in *Proc. of CAV Workshop*, Rutgers NJ, June 1990.
- [16] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, A. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines using BDD's", in *Proc. of ICCAD'90*, November 1990.