The Implicit Set Paradigm:
A New Approach to
Finite State System Verification

O. Coudert and J. C. Madre
Bull Corporate Research Center
Rue Jean Jaurès
78340 Les Clayes-sous-bois, FRANCE

Abstract

This paper presents a new state of the art in the field of finite state system verification. The paradigm of this approach is to represent and to manipulate these systems in an implicit way. The computational costs of the verification procedures using this paradigm depend on the costs of the operations performed on this implicit representation instead of the number of states and transitions of the verified systems. This paradigm allows these new verification procedures to overcome the limitations of previously available techniques.

1 Introduction

We have introduced in 1989 a technique for verifying digital sequential circuits that can handle circuits with state-transition graphs too large to be built. The verification procedure presented in [24, 25], included in a tool called SIAM, consists in traversing the state transition graph of a sequential circuit in a breadth-first manner without building this graph. During the traversal, the procedure checks all the states and all the transitions so it performs exhaustive verifications.

The key idea that allowed SIAM to overcome the limitations of previously developed verification procedures is that it does not manipulate the states and the transitions of this graph explicitly. SIAM manipulates sets of states and sets of transitions that are denoted by their characteristic functions, and these Boolean functions are represented with binary decision diagrams (BDDs) that are a very compact graph representation of such functions [12]. The reason that explains why SIAM can handle circuits that were out of reach of previously available verification procedures is that there is *no relation* between the sizes of the BDDs that SIAM operates on, and the sizes of the sets of states and of transitions that these BDDs denote, so that the computation cost of the procedure is *completely independent* from the number of states and of transitions of the circuit to be verified.

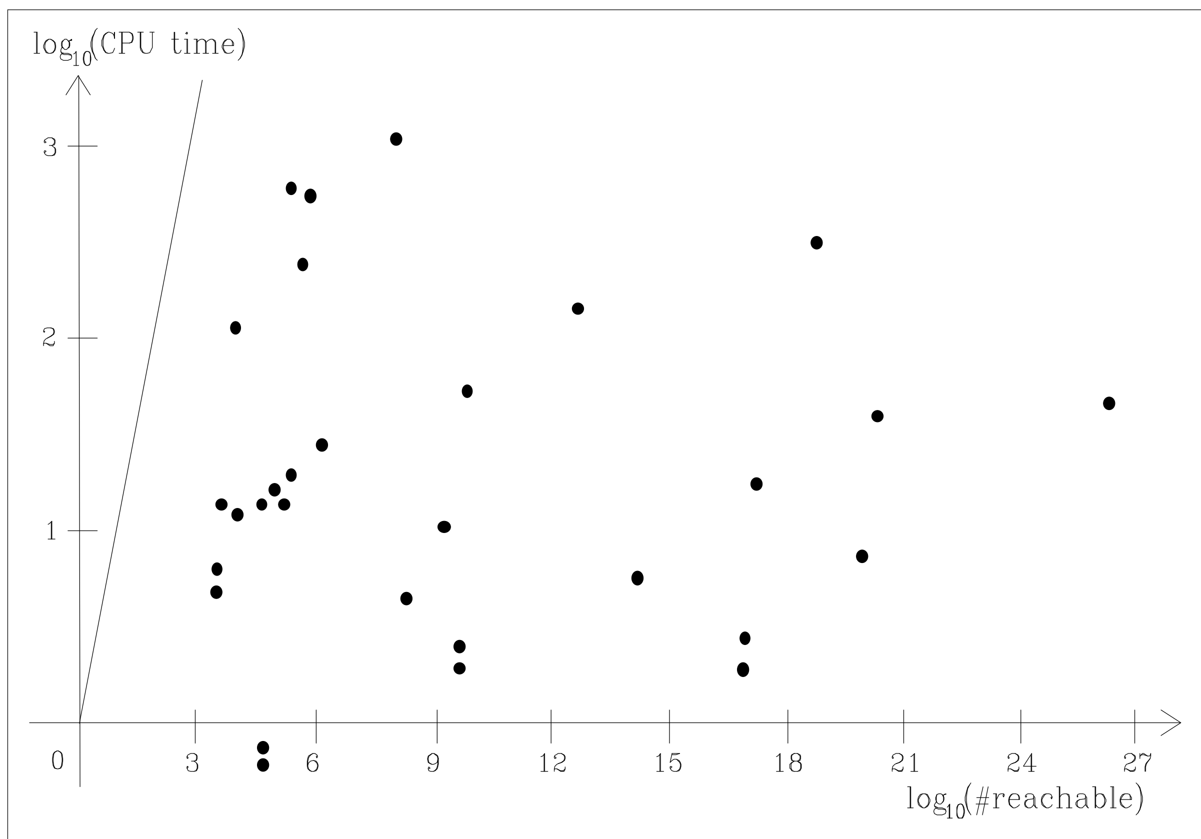

Figure 1. Implicit Reachable State Computation.

This is clearly demonstrated by Figure 1 that shows the CPU times needed to compute the set of reachable states of about 30 different sequential circuits using the BDD based breadth first traversal procedure presented in [24]. On the x-axis of this graph are the logarithms of the numbers of reachable states of the circuits, and on the y-axis are the logarithms of the CPU times in seconds, obtained on a Sun Sparc1 computer, needed to compute the BDDs representing these sets of states.

The straight line on the left corresponds to the CPU times that would be obtained using a procedure with a cost linear with respect to the numbers of reachables states to be computed. This line represents the best results that could be obtained with a procedure manipulating states explicitly. The drawing shows that all computation times are sub-linear with respect to the number of reachable states. There are circuits that are clearly out of reach of procedures manipulating states explicitly. Finally, the regression coefficient for these results is almost equal to 0, which corroborates the theoretical result of independence [28] of the numbers of reachable states of the circuits and the times needed to compute these states.

Since 1989 many efforts have been spent to use the implicit set manipulation paradigm for solving other finite state system verification problems. So far, these efforts have demonstrated that this paradigm applies successfully to several very different problems, including the sequential circuit verification, the verification of dynamic properties of finite state sys-

tems and of synchronous programs, the proof of language containment for ω-automata, and the equivalence of systems modulo the bisimulation.

This paper presents the principles of this paradigm, and its applications so far. Section 2 presents the Quantified Propositional Logic, that is a very simple extention of Propositional Logic sufficient to describe the elementary operations that are used to build the finite state system verification algorithms mentioned above. Section 3 recalls the main properties of the binary decision diagram representation, and gives the complexity of the evaluation on BDDs of formulas of the Quantified Propositional Logic. Section 4 presents the elementary set operations used by the verification algorithms, and explains why so much work has been done to optimize the evaluation of the *image* and *reverse image* operations. In order to apply the verification paradigm presented in this paper, the system to be verified must be described by a set of Boolean equations. Section 5 presents such a model and briefly explains how this model can be produced from different formalisms that are used worldwide to describe and to program finite state systems.

2 Quantified Propositional Logic

In this paper we will use the Quantified Propositional Logic to specify the implicit set operations that are used in the paradigm presented here. Adding quantifiers to Propositional Logic does not increase its expressive power, but it allows us to write very concisely expressions whose quantifier-free forms have exponential sizes. This section presents the syntax and the semantics of the Quantified Propositional Logic.

Quantified propositional formulas are built out of an countable set $\mathcal{V}$ of variables that will be noted $x, x_1, x_2, \ldots, y, y_1, y_2, \ldots, z, \ldots$, the constants $\{0, 1\}$, and the connectors $\{\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, \oplus, \exists, \forall\}$. The syntax of these formulas [44] is defined by the following rules:

$$
\begin{array}{rcl}
<\mathit{QPF}> & ::= & <\mathit{var}> \mid 0 \mid 1 \\
 & ::= & (<\mathit{QPF}>) \\
 & ::= & <\mathit{op_1}><\mathit{QPF}> \\
 & ::= & <\mathit{QPF}><\mathit{op_2}><\mathit{QPF}> \\
 & ::= & <\mathit{quantifier}><\mathit{var}><\mathit{QPF}> \\
<\mathit{op_1}> & ::= & \neg \\
<\mathit{op_2}> & ::= & \vee \mid \wedge \mid \Rightarrow \mid \Leftrightarrow \mid \oplus \\
<\mathit{var}> & ::= & x \mid x_1 \mid x_2 \mid \ldots \\
<\mathit{quantifier}> & ::= & \exists \mid \forall
\end{array}
$$

We note $\mathcal{F}$ the set of formulas that can be built using the preceding rules. A formula that contains no occurrence of the symbols $\exists$ and $\forall$ is said to be quantifier free.

An *interpretation* is a function from the set $\mathcal{V}$ into the set $\{0, 1\}$. Any interpretation i can be extended into a function i_* from the set $\mathcal{F}$ into the set $\{0, 1\}$ using the following rules: $i_*(0) = 0$; $i_*(1) = 1$; if x is a variable, then $i_*(x) = i(x)$; $i_*(\neg f) = 1$ if and only if $i_*(f) = 0$; $i_*(f_1 \vee f_2) = 1$ if and only if $i_*(f_1) = 1$ or $i_*(f_2) = 1$; $i_*(\exists\, x f(x)) = 1$ if and

only if $i_*(f(0)) = 1$ or $i_*(f(1)) = 1$, in addition with the following rewriting rules [28]:

$$\begin{array}{rcl}(\forall x)\ f & \rightarrow & \neg(\exists x\ \neg f) \\ f \wedge g & \rightarrow & \neg(\neg f \vee \neg g) \\ f \Rightarrow g & \rightarrow & \neg f \vee g \\ f \Leftrightarrow g & \rightarrow & (f \Rightarrow g) \wedge (g \Rightarrow f) \\ f \oplus g & \rightarrow & \neg(f \Leftrightarrow g)\end{array}$$

A formula f maps any interpretation i onto a unique element of $\{0,1\}$, and thus it defines a unique function, noted $\lambda i.i_*(f)$, from the set $(\{0,1\} \rightarrow \mathcal{V})$ into the set $\{0,1\}$. Given a total ordering for the variables in $\mathcal{V}$, the formula f defines a unique function from $\{0,1\}^{|\mathcal{V}|}$ into $\{0,1\}$. The formulas f and g are said to be equivalent if and only if the functions they denote are equal. A formula f that is equivalent to 1 is called an tautology, and a formula that is equivalent to 0 is called an antilogy. In the following we will use the same notation for representing a formula and the function it denotes.

A Boolean function is a function from $\{0,1\}^n$ into $\{0,1\}$. The preceding definitions show that any propositional formula f denotes a unique Boolean function from $\{0,1\}^{|\mathcal{V}|}$ into $\{0,1\}$. Conversely any Boolean function from $\{0,1\}^n$ into $\{0,1\}$ can be represented with at least one quantifier free propositional formula built out the set of variables $\{x_1, x_2, \ldots, x_n\}$. This means that any quantified propositional formula is equivalent to at least one quantifier free propositional formula. This quantifier free propositional formula can be obtained with the following rewriting rules [28]:

$$\begin{array}{rcl}(\exists x\ f) & \rightarrow & (f[x \leftarrow 0] \vee f[x \leftarrow 1]) \\ (\forall x\ f) & \rightarrow & (f[x \leftarrow 0] \wedge f[x \leftarrow 1])\end{array}$$

where f is a quantifier free formula, and $f[x \leftarrow g]$ is the formula obtained by replacing all occurrences of x in f with g. We will say in the sequel that computing a *quantifier free form* of a quantified propositional formula consists in eliminating the quantified variables from this quantified formula.

3 Binary Decision Diagrams

Shannon decomposition of a propositional formula f, built out of n variables, with respect to one of its variable x is the unique couple of functions denoted by the couple of formulas $(f[x \leftarrow 0], f[x \leftarrow 1])$ [1]. When iterated for all the variables of f, the decomposition process defined above associates the formula f with a tree that is unique modulo the decomposition ordering, and that is called the *Shannon tree* of f. This tree [1] has $2^n - 1$ vertices and its leaves are the constants 0 and 1.

Figure 2 shows the Shannon tree of the formula $(x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$. The value of the function for any assignment for its variables can be found by following the path, from the root of the tree to a leaf, that this assignment defines in the tree. At each vertex, this path follows the left branch if the associated variable is assigned the value 0, and the right branch if the variable is assigned the value 1. The path defined by the assignment $x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0$ is marked with a dotted line in Figure 2.

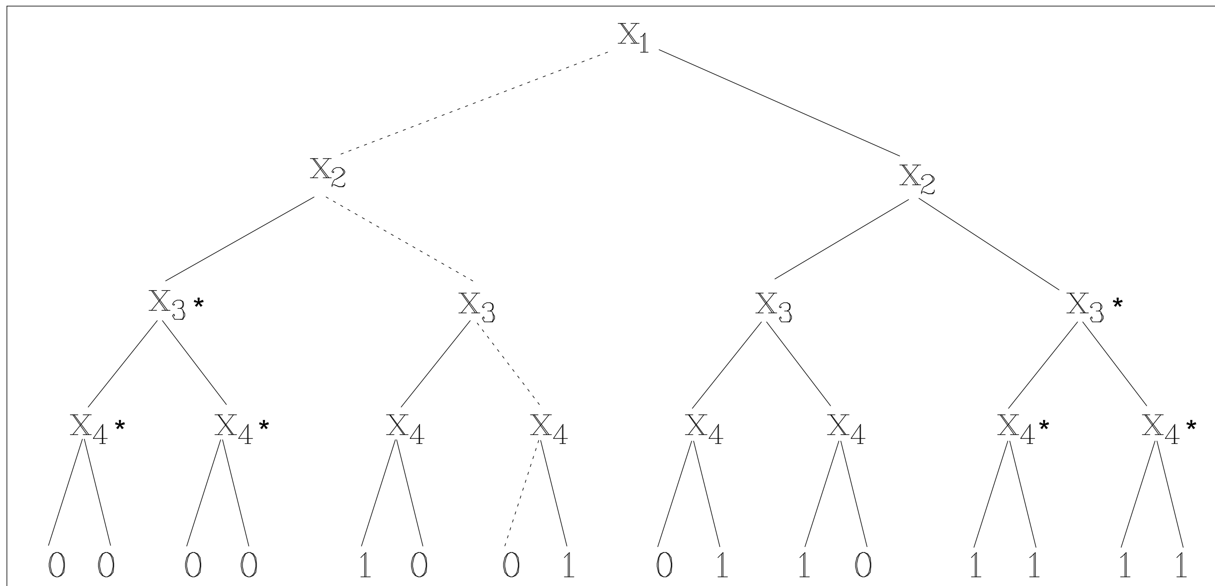

Figure 1. Shannon tree of the formula $(x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$.

The Binary Decision Diagram (BDD) of a formula f is the compacted representation of the Shannon tree of f obtained for a given variable ordering. This BDD is produced by iteratively applying two compaction rules on the Shannon tree [12]. The first compaction rule consists in eliminating any vertex of the tree that has isomorphic sons because this vertex does not contain any valuable information about the function. For instance the vertices marked with a $(*)$ in Figure 2 are useless. The second compaction rule consists in identifying all isomorphic subtrees, so that the tree becomes an acyclic directed graph called the BDD of f. The BDD of the formula $(x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$ is shown by Figure 3.

Both compaction rules participate in making the number of vertices in the BDDs much less than the number of vertices in the associated Shannon trees. In the sequel we note $|f|$ the size of the BDD of the function f, that is the number of vertices in its BDD built with a given variable ordering. Though theoretical results show that the first compaction rule is more effective than the second one [28], the latter is sometimes essential for getting BDDs of tractable sizes. For instance, there are symmetric functions for which application of the first rule alone does not reduce the size of Shannon tree, and whose BDD has a quadratic size for all variable orderings [28]. However there exist BDDs with n variables that have a number of vertices in $O(2^n/n)$, which means that there exist functions that do not have tractable BDDs for a given variable ordering [28].

The structure and the size of the binary decision diagram of a Boolean function can heavily depend on the input variable ordering that has been chosen to build this graph [12, 33]. There exist Boolean functions of $2n$ variables that have a BDD with a size in $O(n)$ for one variable ordering and a size in $O(2^n)$ for another variable ordering. Moreover, there exist functions that do not have tractable BDDs for any variable ordering [13]. There is an algorithm for computing the best variable ordering for a function, that is an ordering

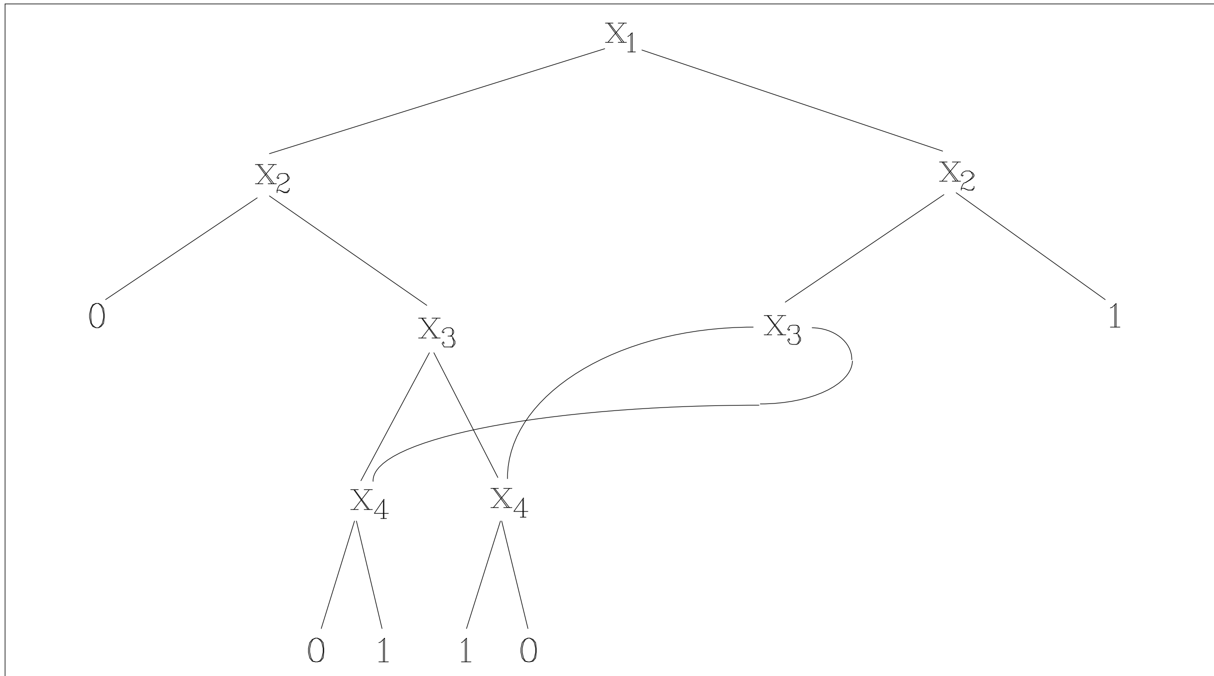

Figure 1. BDD of the formula $(x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$.

for which $|f|$ is minimal [33]. However this algorithm has a complexity in $O(n^2 3^n)$, which makes it unusable for complex functions.

The usual Boolean operators can be evaluated with a polynomial complexity on BDDs built with the same variable ordering [11, 12]. For any functions f and g, the time and memory needed to compute the BDD of the function $(f \star g)$, where $\star$ is one of the operators $\{\vee, \wedge, \Rightarrow, \Leftrightarrow, \oplus\}$ is in $O(|f| \times |g|)$. The complexity is the same for all these operators because all combinations can be performed using a unique operator called "apply" that has the mentioned complexity. This is a remarkable property that makes BDDs very different from previously used representations of propositional formulas, for instance the sum-of-products representation for which the operators have different complexities, most of them being at least exponential [28].

Building the BDD of the function that is denoted by the quantified propositional formulas $(\exists x f)$ and $(\forall x f)$, where f is a quantifier free formula, can be done in $O(|f|^2)$ [28]. Indeed this quantified variable elimination basically consists in combining the formulas $f[x \leftarrow 0]$ and $f[x \leftarrow 1]$ using either the disjunction or the conjunction. When there are several variables to be eliminated the quadratic complexity of each elimination can make the complete elimination exponential. There exist quantified propositional formulas of the form $(\exists x_1 x_2 \ldots x_n\ f)$ whose quantifier free form has a BDD whose size is exponential with respect to n or/and the size of the BDD of f [28].

The situation is very similar with the substitution operation. Computing the BDD of the propositional formula $f[x \leftarrow g]$ from the BDDs of the formulas f and g respectively can be done in $O(|f|^2 \times |g|)$. The question is open to know whether this can be done in

$O(|f| \times |g|)$ [12]. However there exist formulas f and $g_1, \ldots, g_n$ with $2n$ variables, such that $(|g| + \sum_{k=1}^{n} |f_k|)$ is in $O(n)$, and such that $|f(g_1, \ldots, g_n)|$ is in $O(2^n)$ [28]. This means that the substitution (or composition) operation of n variables is exponential with respect to n in the worst case [28].

Despite these obvious limitations, binary decision diagrams currently are the representation of propositional formulas and of Boolean functions that support the most efficiently the evaluation of the operators of the Quantified Propositional Logic defined in Section 2. For this reason, all the verification procedures using the implicit set paradigm represent the Boolean sets they manipulate with Binary Decision Diagrams.

4 Implicit Boolean Set Manipulation

This section explains why binary decision diagrams are a good representation for manipulating Boolean sets, and it gives the complexities of the elementary set operators.

A Boolean function f from $\{0,1\}^n$ into $\{0,1\}$ denotes a unique subset S_f of $\{0,1\}^n$ that is defined by the equation

$$S_f = \{x \in \{0,1\}^n \ / \ f(x) = 1\}.$$

Conversely a subset S of $\{0,1\}^n$ is denoted by a unique Boolean function from $\{0,1\}^n$ into $\{0,1\}$, noted χ_S, that is defined by the equation

$$\chi_S(\vec{x}) = 1 \quad \text{if and only if} \quad \vec{x} \in S,$$

and is called its *characteristic function*. Since any Boolean function from $\{0,1\}^n$ into $\{0,1\}$ has a unique BDD representation for a given variable ordering, any subset of $\{0,1\}^n$ also has a unique BDD for this variable ordering [28].

Characteristic functions are a very interesting representation of Boolean sets because there is no relation at all between the number of elements in a set and the size of the BDD that denotes this set, so that huge Boolean sets can potentially be denoted by small BDDs [24]. There exist of course subsets of $\{0,1\}^n$ whose BDD has an exponential size with respect to n. Boolean operators correspond with set operators, for instance disjunction corresponds with union, and negation with complementation. All elementary set operations can thus be evaluated with a quadratic complexity on BDDs. Moreover, thanks to this correspondence between Boolean sets and Boolean functions, any set operation over such sets can be expressed using the Quantified Propositional Logic defined in Section 2 [28].

Image and *reverse image* computations of functions from the set $\{0,1\}^n$ into the set $\{0,1\}^m$ can be expressed in terms of the substitution and the variable elimination operations defined in Section 2 [27]. A function from $\{0,1\}^n$ into the set $\{0,1\}^m$ is defined by a functional vector $\vec{f} = [f_1 \cdots f_m]$ of Boolean functions. For any subset S of $\{0,1\}^n$, the characteristic function of the set $\{\vec{y} \in \{0,1\}^m \ / \ \exists \vec{x} \in S, \ \vec{f}(\vec{x}) = \vec{y}\}$, noted $Img(\vec{f}, \chi_S)$, is defined by the equation [25]:

$$Img(\vec{f}, \chi_S) \quad = \quad \lambda\vec{y}.(\exists\vec{x} \ \chi_S(\vec{x}) \wedge \bigwedge_{k=1}^{m} f_k(\vec{x}) \Leftrightarrow y_k).$$

Conversely, for any subset S of $\{0,1\}^m$, the characteristic function of the reverse image of S, which is the set $\{\vec{x} \in \{0,1\}^n \ / \ \vec{f}(\vec{x}) \in S\}$, is noted $Rev(\vec{f}, \chi_S)$, and is defined by the equation [26]:

$$Rev(\vec{f}, \chi_S) \quad = \quad \lambda\vec{x}.(\chi_S(f_1(\vec{x}), \ldots, f_m(\vec{x}))).$$

A relation R over the set $\{0,1\}^n \times \{0,1\}^m$ is a function from $\{0,1\}^n \times \{0,1\}^m$ into $\{0,1\}$. The first projection of a subset S of $\{0,1\}^n \times \{0,1\}^m$ with respect to R, noted $First(R, S)$, is the characteristic function of the set $\{\vec{x} \in \{0,1\}^n \ / \ \exists\vec{y} \in \{0,1\}^m, \ (\vec{x}, \vec{y}) \in S \wedge R(\vec{x}, \vec{y}) = 1\}$, and is defined by the equation [15]:

$$First(R, \chi_S) \quad = \quad \lambda\vec{x}.(\exists\vec{y} \ \chi_S(\vec{x}, \vec{y}) \wedge R(\vec{x}, \vec{y})).$$

In a similar way the second projection of S with respect to R, noted $Second(R, S)$, is the characterictic function of the set $\{\vec{y} \in \{0,1\}^m \ / \ \exists\vec{x} \in \{0,1\}^n, \ (\vec{x}, \vec{y}) \in S \ \wedge \ R(\vec{x}, \vec{y}) = 1\}$, and is defined by the equation [15]:

$$Second(R, \chi_S) \quad = \quad \lambda\vec{y}.(\exists\vec{x} \ \chi_S(\vec{x}, \vec{y}) \wedge R(\vec{x}, \vec{y})).$$

Quantified variable elimination and variable substitution operations both have exponential complexities with respect to the size of the BDDs in the worst case, so the operators *Img*, *Rev*, *First*, and *Second* defined above also have this exponential complexity. This means that these operators are much more costly than the elementary set operators, and that the complexity of any algorithm that uses these operators have an exponential complexity with respect to the size of the BDDs [28]. This explains why many efforts have been put on optimizing the evaluation of these operations.

Some optimizations take advantage of the structural properties of BDDs, for instance the image computation algorithms for functional vectors presented in [24, 25, 20, 43] (based on the *constrain* operator [25], also called *generalized cofactor and image restrictor*), and the reverse image algorithms presented in [26, 54]. Some other optimizations take advantage of the properties of the logical operators occurring in the formulas to be evaluated in order to reduce as much as possible the sizes of the BDDs that must be manipulated during the these operations. Such an optimization that can be used for computing images of functional vectors is presented in [59] and similar ones were later presented in [17, 37]. Finally some optimizations consist in taking into account the context in which these expressions must be evaluated to either reduce the sizes of the manipulated BDDs [26, 16, 56], or to reduce the number of iterations performed during the verifications with the *iterative squaring* technique [15] or with a mix of depth-first and breadth-first traversal [35].

5 Finite State System Verification

In order to reason about finite state systems using the implicit Boolean set operations presented in Section 4 it is necessary to have a representation of this system in which objects to be treated are Boolean sets. In this section we present two closely related representations that fit this purpose and briefly review the techniques that have been

proposed to produce these representations from formalisms as different as the VHDL hardware description language, the synchronous languages LUSTRE or ESTEREL, and the CCS process calculus.

A Finite State Machine (FSM) $\mathcal{M}$ can be defined in two equivalent ways that differ on the way the outputs and transitions are specified. A FSM model using functional vectors consists of a 6-tuple $(n, m, r, \vec{\omega}, \vec{\delta}, \mathit{Init})$ [27, 39], and a model using relations consists of a 6-tuple $(n, m, r, \Omega, \Delta, \mathit{Init})$ [15], where:

- n is the number of Boolean state variables of $\mathcal{M}$. The state space of the machine is thus $\{0,1\}^n$.
- m is the number of Boolean inputs of $\mathcal{M}$. The input space of the machine is thus equal to $\{0,1\}^m$.
- The functional vector $\vec{\omega}$ is built out of the r output functions of $\mathcal{M}$. These functions can be either partially or completely defined [27]. In the first case each function of the functional vector $\vec{\omega}$ is represented with a couple of functions (f, C) where f is a function from $\{0,1\}^n \times \{0,1\}^m$ into $\{0,1\}$ and C is the characteritic function of the subset of $\{0,1\}^n \times \{0,1\}^m$, sometimes called the care set of f [45], on which the function is defined. In the second case each of the functions in $\vec{\omega}$ is a function from $\{0,1\}^n \times \{0,1\}^m$ into $\{0,1\}$. The output relation Ω is defined on $\{0,1\}^n \times \{0,1\}^m \times \{0,1\}^r$, and $\Omega(\vec{y}, \vec{x}, \vec{z}) = 1$ if and only if the machine produces the output $\vec{z}$ when it reads the input $\vec{x}$ while it is in the state $\vec{y}$.
- The functional vector $\vec{\delta}$ is built out of the n next-state or transition functions of $\mathcal{M}$. The transition function $\vec{\delta}$ can be either partially or completely defined. In the first case, we consider the care set of this vectorial function to be the intersection of all the care sets of the functions in $\vec{\delta}$, so that its characteristic function C is equal to $C = \bigwedge_{k=1}^{n} C_k$. The transition relation Δ is defined on $\{0,1\}^n \times \{0,1\}^m \times \{0,1\}^n$ and $\Delta(\vec{y}, \vec{x}, \vec{y'}) = 1$ if and only if the machine goes into the state denoted by $\vec{y'}$ when it reads the input $\vec{x}$ while it is in the state $\vec{y}$.
- Init is the characteristic function of the set of initial states of the machine $\mathcal{M}$.

Gate level descriptions of digital synchronous circuits written in a format such as BLIF are representations of this kind [62]. A circuit description written in BLIF is made of the interface part describing the interface of the circuit, the declaration part giving the names of its binary storage elements, and a part describing the multi-level logic network that computes the outputs and the next values of the state variables from the current values of these variables and the values of the inputs. The care set of the output and the transition functions can be specified.

The main problem that has to be solved to get BDD representations of tractable sizes from BLIF descriptions is to compute good variable orderings, and many efforts have been spent to solve this problem. Since the algorithm for computing the best variable ordering for a function has a much too high complexity to be used on non trivial descriptions, efforts have been mainly dedicated to develop ordering heuristics and reordering heuristics.

Several proposed ordering heuristics exploit the structure of the multi-level logic network to produce the input and state variable ordering [3, 34, 50, 51, 52].

Others heuristics produce this variable ordering from simple or more complex analysis of the supports of the output and the transition functions [43, 59]. A special kind of BDDs have even been developed that give a way to evaluate very efficiently different variable orderings in order to choose the best one [18]. Since these heuristics sometimes fail to produce a good variable ordering, reordering heuristics have been developed to reduce the size of existing BDDs. These heuristics are all based on cost functions that measure the gain obtained by exchanging the ordering of two variables [19, 41, 42].

Compilation processes have been developed to produce FSM models from reactive programs written in LUSTRE and ESTEREL. LUSTRE is a declarative language that allows programmers to express relations between the synchronous input and output flows of a reactive system [36]. The FSM model compilation for this language consists in determining the set of state variables that are necessary to produce, at each instant, the values of the outputs from the current and past values of the inputs [58, 56]. There are LUSTRE programs that cannot be compiled into a FSM model, for instance programs that manipulate real numbers, but all programs that operate on variables of finite domains can be processed. ESTEREL is an imperative language that allows engineers to program reactive systems that have very complex control structures [6]. The FSM model compilation process for ESTEREL, that currently handles ESTEREL programs operating on Boolean variables, consists in determining the state variables that are necessary to implement the control structure of the programs [4, 5]. Both compilation processes generate multi-level logic networks on which the heuristics mentioned above can be applied.

FSM model compilation processes for hardware description languages actually are logic synthesis processes. Such processes have been developed for several hardware description languages, for instance LDS [48, 49] that is a hardware description language used at Bull to describe circuits at the register transfer level, and for VHDL [23, 29], that is the IEEE standard language for describing circuits. These processes rely on different basic techniques such as symbolic execution [8], state machine encoding, and state machine minimisation [7, 46]. The main problem here is to find state encodings and variable orderings that minimize the BDDs that represent the FSM and the BDDs manipulated during the verifications.

Finally FSM model compilation processes have been defined for different process calculi and labelled transition systems [10, 32]. The first problem to be solved here is to define the composition rules of FSM models that correspond to the different operators of these formalisms. The second problem is the same as the one mentioned in the previous paragraph, which is to produce BDDs of tractable size. Work has been done to define composition rules for variable orderings that correspond with the different composition operators, for instance the sequential and parallel composition operators [32]. The problem with these rules is that they are static so that, experience demonstrates it, the attempts that are made to have a FSM model of moderate size do not prevent the BDDs of the sets manipulated during the verification to grow exponentially.

From the historical point of view, the first verification problem that has been tackled

using the paradigm presented here has been to prove the equality or the inclusion of the languages produced by two finite state machines [24]. In parallel several verification procedures [15, 9, 26] have been developed to check properties expressed in the branching temporal logic [21, 22, 31, 55]. Experience has shown that these verification techniques can handle finite state machines that could not be treated with previously available verification techniques, e.g. [30, 40, 57, 22, 38, 55], because their state transition diagrams is much too large to be built, or because their states are too many to be explicitly enumerated. For instance these BDD based techniques has been successfully used to study the cache consistency protocol of a multi processor computer [47].

Several optimizations have been later proposed to improve the fixpoint computations that are needed for these verification tasks [35, 53, 59, 61]. Specialized procedures have been developed to check dynamic properties of digital circuits [9, 14], and of programs written in the synchronous language LUSTRE. Also procedures have been developed to check the containment for languages produced by ω-automata [15, 60], and the equivalence of systems with respect to the bisimulation [15, 10].

All these verification methods use fixpoint computations that consist in virtually traversing the state diagram of a finite state machine breadth-first in a forward [24] or backward way [15, 26]. These fixpoint computations built convergent series of sets that are represented by BDDs. Though the equations that define these sets are different for each problem, they all make use of one or several of the operations *Img*, *Rev*, *First*, and *Second*, and the difference in performance between procedures that address the same problem comes from the optimizations used to evaluate these operations.

6 Conclusion

In this paper, we have presented the implicit set approach to finite system verification. Many different verification procedures based on this approach have shown that the implicit Boolean set manipulations presented here are a robust basis for implementing finite system verification techniques that can handle problems that are out of reach of previously available techniques.

There are of course cases where the implicit set manipulation paradigm is not sufficient to allow the verification procedures to tackle the problems. The computation costs of these procedures depend only on the sizes of the BDDs they manipulate, so these procedures fail on certain systems because these BDDs become too large. Experience shows that there are finite state machines whose transition and output functions have very small BDDs, namely less than 100 vertices each, and for which it is impossible to compute the BDD that represents their sets of reachable states, because the sizes of the BDDs that are successively generated during the fixpoint computation grow exponentially.

The explanation for such a behaviour is that the variable ordering used to build these BDDs is not a good variable ordering. The efforts put on variable ordering heuristics have provided us with means of getting FSM models with tractable sizes, but this size reduction only concerns the BDDs of the transition and output functions of the machine. There is *no reason* why this static reduction should help the BDDs generated during the forward

or backward traversals of the state transition graphs of the machines to stay small. This is a strong argument for the development of fast reordering heuristics that would allow us to rebuild all the manipulated BDDs during the fixpoint computations. The problem is that experience shows that in some cases the variable ordering for which the BDDs of the transition and output functions of the FSM are small, and the variable ordering for which the BDDs in the fixpoint computation are small, are not *compatible*. The problem then is to determine whether it is possible to generate with a moderate cost, a different finite state machine with an isomorphic state transition graph, i.e. a FSM in which the states are encoded differently, for which the transition and output functions have small BDDs, and for which the particular fixpoint computation to be performed also generates small BDDs.

It appears that a bottleneck of formal verification, despite the improvements of the implicit traversal techniques, consists in both expressing the properties that one wants to check, and modeling the system that one wants to validate. On one hand, several languages have been developed to describe complex temporal behaviors, but there is still work to do before any user can legibly express the properties he expects from its design. On the other hand sequential systems that are too complex to be directly treated by BDD based techniques require a rewriting step that consists in abstracting parts of its behavior, which is not an easy task because its automation is limited.

References

[1] S. B. Akers, “Binary Decision Diagrams”, *IEEE Transactions on Computers*, Vol C-27, N°6, 1978.

[2] P. Ashar, A. Ghosh, S. Devadas, A. R. Newton, “Combinational and Sequential Logic Verification Using General Binary Decision Diagrams”, in Proc. of *International Workshop on Logic Synthesis*, MCNC North Carolina, May 1991.

[3] C. L. Berman, “Ordered Binary Decision Diagrams and Circuit Structure”, in Proc. of *International Conference on Computer Design'89*, Cambridge, Massachusetts, September 1989.

[4] G. Berry, “A Hardware Implementation of ESTEREL”, in Proc. of *1991 International Workshop on Formal Methods in VLSI Design*, Miami Florida, January 1991.

[5] G. Berry, “ESTEREL on Hardware”, in Proc. of *The Royal Society Discussion Meeting on Mechanized Reasoning and Hardware Design*, London, UK, October 1991.

[6] G. Berry *et al.*, “The ESTEREL Language”, in *IEEE Special Issue on Synchronous Languages*, September 1992.

[7] C. Berthet, O. Coudert, J. C. Madre, “New Ideas on Symbolic Manipulations of Finite State Machines”, in Proc. of *International Conference on Computer Design'90*, Cambridge, Massachusetts, September 1990.

[8] J. P. Billon, “Symbolic Execution of Discrete Programs”, *BULL Research Report N°87039*, September 1987.

[9] S. Bose, A. Fisher, "Automatic Verification of Synchronous Circuits Using Symbolic Logic Simulation and Temporal Logic", in *Formal VLSI Correctness Verification*, L.J.M. Claesen Editor, North-Holland, pp. 151–158, November 1989.

[10] A. Bouali, R. de Simone, "Symbolic Bisimulation Minimisation", submitted to *Computer Aided Verification Workshop'92*, Montreal, Canada, 1992.

[11] K. S. Brace, R. L. Rudell, R. E. Bryant, "Efficient Implementation of a BDD Package", in Proc. of *27th Design Automation Conference*, Orlando, Florida, June 1990.

[12] R. E. Bryant, "Graph-Based Algorithms for Boolean Functions Manipulation", *IEEE Transactions on Computers*, Vol C35, $N^0$8, pp. 677–692, August 1986.

[13] R. E. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication", Carnegie Mellon University Research Report, September 1988.

[14] R. E. Bryant, D. L. Beatty, C-J. H. Seger, "Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation", in Proc. of *28th Design Automation Conference*, pp. 397–402, San Francisco, California, June 1991.

[15] S. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond", in Proc. of *Logic In Computer Science*, Philadelphia, June 1990.

[16] S. Burch, E. M. Clarke, K. L. McMillan, "Sequential Circuit Verification Using Symbolic Model Checking", in Proc. of *27th Design Automation Conference*, Orlando, Florida, July 1990.

[17] S. Burch, E. M. Clarke, D. E. Long, "Representing Circuits More Efficiently in Symbolic Model Checking", in Proc. of *28th Design Automation Conference*, pp. 403–407, San Francisco, California, June 1991.

[18] K. M. Butler, D. E. Ross, R. Kapur, M. R. Mercer, "Heuristics to Compute Variable Orderings for Efficient Manipulations of Ordered Binary Decision Diagrams", in Proc. of *28th Design Automation Conference*, pp. 417–420, San Francisco, California, June 1991.

[19] N. Calazans, R. Jacobi, Q. Zhang, C. Trullemans, "Improving BDD manipulation through Incremental Reduction and Enhanced Heuristics", in Proc. of *CICC'91*, May 1991.

[20] H. Cho, G. D. Hachtel, S. W. Jeong, B. Plessier, E. Schwarz, F. Somenzi, "ATPG Aspect of FSM Verification", in Proc. of *IEEE International Conference on Computer Aided Design'90*, Santa Clara, California, November 1990.

[21] E. M. Clarke, E. A. Emerson, A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Verification", in ACM Trans. Programming Language System, Vol. 8, No. 2, 1986.

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

[23] D. R. Coelho, *The VHDL Handbook*, Kluwer Academic Publishers, 1989.

[24] 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.

[25] O. Coudert, C. Berthet, J. C. Madre, "Verification of Sequential Machines using Boolean Functional Vectors", in *Formal VLSI Correctness Verification*, L.J.M. Claesen Editor, North-Holland, pp. 179–196, November 1989.

[26] O. Coudert, J. C. Madre, "Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams", in *Computer-Aided Verification'90*, E. M. Clarke and R. P. Kurshan Editors, DIMACS Series, pp. 75–84, June 1990.

[27] O. Coudert, J. C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits", in Proc. of *IEEE International Conference on Computer Aided Design'90*, Santa Clara, California, November 1990.

[28] O. Coudert, *SIAM : Une Boite à Outils pour la Preuve Formelle de Systèmes Séquentiels*, PhD. Thesis, Ecole Nationale Supérieure des Télécom1munications, Paris, France, October 1991.

[29] A. Debreil, C. Berthet, A. Jerraya, "Symbolic Computation of VHDL Hierarchical Descriptions", in Proc. of the *First European Conference on VHDL Methods*, Marseilles, France, September 1990.

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

[31] E. A. Emerson, "Temporal and Modal Logic", *Formal Models and Semantics*, Handbook of Theorical Computer Science, Jan van Leeuwen Editor, Elsevier, pp. 995–1072, 1990.

[32] R. Enders, T. Filkorn, D. Taubner, "Generating BDDs for Symbolic Model Checking in CCS", in Proc. of the *Computer Aided Verification Workshop'91*, Aalborg, Denmark, 1991.

[33] S. J. Friedman, K. J. Supowit, "Finding the Optimal Variable Ordering for Binary Decision Diagrams", *IEEE Transactions on Computer, Vol C-39, N° 5*, pp. 710–713, May 1990.

[34] M. Fujita, H. Fujisawa, N. Kawato, "Evaluation and Improvements of Boolean Comparison Methods Based on Binary Decision Diagrams", Proc. of *IEEE International Conference on Computer Aided Design'88*, Santa Clara, California, November 1988.

[35] A. Ghosh, S. Devadas, "A Mixed Depth-First/Breadth-First Traversal Technique for Sequential Logic Verification", in Proc. of the *International Workshop on Logic Synthesis*, MCNC, North Carolina, May 1991.

[36] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, "The Synchronous Data-Flow Programming Language LUSTRE", in *IEEE Special Issue on Synchronous Languages*, pp. 1305–1320, September 1992.

[37] H. Hiraishi, K. Hamaguchi, H. Ochi, S. Yajima, "Vectorized Symbolic Model Checking of Computation Tree Logic for Sequential machine Verification", in Proc. of the *Computer Aided Verification Workshop'91*, Aalborg, Denmark, 1991.

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

[39] J. E. Hopcroft, J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, Massachusetts, 1979.

[40] S. H. Hwang, A. R. Newton, "An efficient Design Correctness Checker of Finite State Machines", in Proc. of *IEEE International Conference on Computer Aided Design'87*, Santa Clara, California, November 1987.

[41] N. Ishiura, H. Sawada, S. Yajima, "Minimization of Binary Decision Diagrams Based on Exchanges of Variables", in proc. of *IEEE International Conference on Computer Aided Design'91*, Santa Clara, California, November 1991.

[42] R. Jacobi, N. Calazans, C. Trullemans, "Incremental Reduction of Binary Decision Diagrams", in Proc. of *ISCAS'91*, June 1991.

[43] S. W. Jeong, B. Plessier, G. D. Hachtel, F. Somenzi, "Variable Ordering for FSM Traversal", in Proc. of the *International Workshop on Logic Synthesis*, MCNC, North Carolina, May 1991.

[44] S. C. Kleene, *Mathematical Logic*, John Wiley and Sons, NY, 1967

[45] B. Lin, H. J. Touati, A. R. Newton, "Don't Care Minimization of Multi-Level Sequential Logic Networks", in Proc. of *IEEE International Conference on Computer Aided Design'90*, Santa Clara, California, November 1990.

[46] B. Lin, "Efficient Symbolic Manipulation of Equivalence Relations and Classes", in Proc. of the *International Workshop on Logic Synthesis*, MCNC, North Carolina, May 1991.

[47] K. L. McMillan, J. Schwalbe, "Formal Verification of the Encore Gigamax Cache Consistency Protocol", in Proc of the *International Symposium on Shared Memory Multiprocessors*, 1991.

[48] J. C. Madre, J. P. Billon, "Proving Circuit Correctness using Formal Comparison Between Expected and Extracted Behaviour", in Proc. of the *25th Design Automation Conference*, Anaheim, California, July 1988.

[49] J. C. Madre, "PRIAM, Un Outil de Preuve Formelle de Circuits Digitaux", Thèse de troisième cycle, Ecole Nationale Supérieure des Télécommunications, Paris, France, June 1990.

[50] S. Malik, A. R. Wang, R. K. Brayton, A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment", in Proc. of *IEEE International Conference on Computer Aided Design'88*, Santa Clara, California, pp. 6–9, November 1988.

[51] S. Minato, N. Ishiura, S. Yajima, "Fast Tautology Checking Using Shared Binary Decision Diagrams - Experimental Results", in *Formal VLSI Correctness Verification*, L.J.M. Claesen Editor, North-Holland, pp. 107–111, November 1989.

[52] S. Minato, N. Ishiura, S. Yajima, "Shared Binary Decision Diagrams with Attributed Edges for Efficient Boolean Function Maniplation", in Proc. of the *27th Design Automation Conference*, Las Vegas, Nevada, pp. 52–57, June 1990.

[53] C. Pixley, "A Computational Theory and Implementation of Sequential Hardware Equivalence", in Proc. of the *Computer Aided Verification'90*, E. M. Clarke and R. P. Kurshan Editors, DIMACS Series, pp. 293–320, June 1990.

[54] C. Pixley, S. W. Jeong, G. D. Hachtel, "Exact Calculation of Synchronization Sequences Based on Binary Decision Diagrams", in Proc. of *29th Design Automation Conference*, Anaheim, California, 1992.

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

[56] C. Ratel, N. Halbwachs, P. Raymond, "Programming and verifying Critical Systems by Means of the Synchronous Dataflow Programming Language LUSTRE", in Proc. of *ACM SigSoft Conference on Software for Critical Systems*, New Orleans, December 1991.

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

[58] G. Thuau, B. Berkane, "Using the Language LUSTRE for Sequential Circuit Verification", in Proc. of the *International Workshop on Designing Correct Circuits*, Lingby, Denmark, January 1992.

[59] 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 *IEEE International Conference on Computer Aided Design'90*, Santa Clara, California, November 1990.

[60] H. J. Touati, R. K. Brayton, R. Kurshan "Testing Language Containment for ω-Automata using BDD's", in Proc. of the *1991 International Workshop on Formal Methods in VLSI Design*, Miami, Florida, January 1991.

[61] F. Van Aelten, J. Allen, S. Devadas, "Verification of Relations between Synchronous Machines", in Proc. of *IEEE International Conference on Computer Aided Design'91*, Santa Clara, California, November 1991.

[62] S. Yang, *Logic Synthesis and Optimization Benchmarks User Guide*, Microelectronics Center of North Carolina, January 1991.