VERIFICATION OF SYNCHRONOUS SEQUENTIAL MACHINES BASED ON SYMBOLIC EXECUTION

Olivier COUDERT, Christian BERTHET, Jean Christophe MADRE

BULL Research Center, P.C. 58B

68 Route de Versailles, 78430 Louveciennes, FRANCE

Abstract

This paper presents an original method to compare two synchronous sequential machines. The method consists in a breadth first traversal of the product machine during which symbolic expressions of its observable behaviour are computed. The method uses *formal manipulations on boolean functions* to avoid the state enumeration and diagram construction. For this purpose, new algorithms on boolean functions represented by Typed Decision Graphs has been defined.

1. Introduction

The VLSI circuits designers must produce *zero-defect circuits* because prototyping is a much too expensive way of debugging circuits [1]. For several reasons, the main of which being the need for high performances, few automated synthesis tools are used. Thus, the emphasis is put on verification techniques. On the other hand, the need for zero-defect design requires a complete and exhaustive verification which cannot be achieved by simulation means because of the combinational complexity.

Formal verification of hardware consists in comparing a circuit realization (hardware device) with a specification (expected behavior). Within BULL's design methodology, circuits are described using the hardware description language LDS [10], which is very similar to procedural VHDL [2]. A behavioral description of a circuit using LDS is a procedural program which computes in a clock cycle the output functions of the circuit and the excitation functions of its storage elements. Several tools have been developed at BULL to extract behavioural descriptions from structural (at layout, transistor or gate levels) descriptions [13]. This extraction process produces an LDS program which translates exactly the functional behavior of the realized circuit. Thus from our point of view, the formal verification of a circuit consists in comparing two LDS programs, the LDS program obtained through the extraction tools, called the realization, and another LDS program, called the specification.

A powerful formal verifier called PRIAM has been designed at BULL's Research Center and developed at BULL's CAD Division [4] [11]. PRIAM can handle industrial circuits such as 64 bits ALU's and other data-path operators with up to 40000 transistors. The core of PRIAM is a powerful theorem

prover in propositional logic [12]. Boolean expressions are represented by Typed Decision Graphs (TDG) [3], the most compact representation currently known.

PRIAM has a severe limitation: it is able to compare a specification program with the realization only if both programs have the same registers and the same state encoding. Frequently this requirement is not satisfied, because the realization and the specification have generally not the same hierarchical structure and the state variables do not coincide. The verification process performed by PRIAM had to be extended to a more general problem: the comparison of two sequential machines, the specification and the realization, each one using its own set of memory elements.

2. Formal Proof of Synchronous Sequential Machines

For the sake of clarity, we study here the problem of equivalence between two completely specified machines. The more general problem of *implication* between two *incompletely specified* machines can be treated in the same way by adapting some definitions.

A *deterministic finite state machine* $\mathcal{M}$ is defined by the 6-tuple $(\Sigma, O, S, s_0, \delta, \lambda)$ where Σ is the input alphabet, O the output alphabet, S the finite set of states, s_0 the initial state, δ the transition function from $S \times \Sigma$ to S, and λ the output function from $S \times \Sigma$ to O [8]. The output sequence $z_1 \ldots z_n$ *generated* by $\mathcal{M}$ from the state s_0 when reading the input sequence $x_1 \ldots x_n$ is such that for $1 \leq k \leq n$, $z_k = \lambda(s_{k-1},x_k)$ and $s_k = \delta(s_{k-1},x_k)$. For two machines $\mathcal{M}_1$ and $\mathcal{M}_2$ operating on the same input and output alphabets, we say that $\mathcal{M}_1$ is *equivalent* to $\mathcal{M}_2$ iff for each input sequence $x_1 \ldots x_n$ of Σ^*, $\mathcal{M}_1$ and $\mathcal{M}_2$ generate the same output sequence $z_1 \ldots z_n$ of O^*.

The usual method to prove the equivalence between the machines $\mathcal{M}_1 = (\Sigma, O, \delta_1, \lambda_1, S_1, s_{01})$ and $\mathcal{M}_2 = (\Sigma, O, \delta_2, \lambda_2, S_2, s_{02})$ consists in building the *product machine* $\mathcal{M} = (\Sigma, \{\text{True, False}\}, \delta, \lambda, S, s_0)$ where we have $S =_{def} S_1 \times S_2$, $s_0 =_{def} (s_{01}, s_{02})$, the output $\lambda((s_1, s_2),x) =_{def} (\lambda_1(s_1, x) \equiv \lambda_2(s_2, x))$ and the transition function $\delta((s_1, s_2), x) =_{def} (\delta_1(s_1, x), \delta_2(s_2, x))$. Then the machines $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent iff for every transition of $\mathcal{M}$ that can be reached from the initial product state (s_{01}, s_{02}), the machine $\mathcal{M}$ produces the output True. This method derives from the one to compare two finite state recognizers [8].

2.1. Breadth First Execution of a Sequential Machine

The explicit construction of the product machine is a very time and memory consuming operation [14]. Several methods have been proposed to enumerate the states of this machine without building it. As far as we know, only the depth first enumeration has been used [6] [9]. In this section, we propose a proof method based on breadth first traversal. We then show how this algorithm supports symbolic manipulations.

The comparison algorithm of two finite state machines using a *breadth first execution* of the product machine is shown in Figure 1. It considers all the states and transitions reachable from the set `M.init` of initial states. If the two machines are equivalent, the execution process stops when no new

product state is reached (statement 7), and the variable `k` then is the length of the longest acyclic path of the product machine starting from an initial state. Note that the proof remains correct if the set of states `From` with which the next set of reached states `Y` is computed contains any already discovered state belonging to `Reached`. So the statement "`From := New`" can be replaced by "**choose** `From` **in such a way that** `New` ⊂ `From` ⊂ `Reached`".

```
function prove(M : Finite-State-Machine) : boolean;
var k : int;
    From, Reached, Y, New : Set-Of-States;
    Z : Set-Of-Outputs;
    X : Set-Of-Inputs;
begin
  k := 0;  New := M.init;  Reached := New;
  do loop
     From := New;                        /* 1: the states to be verified.   */
     X := new-inputs(k);                 /* 2: new set of inputs.           */
     Z := {λ(y, x) / y ∈ From, x ∈ X};   /* 3: compute generated ouputs.    */
     If Z ≠ {True} then return(False);   /* 4: are the ouputs corrects?     */
     Y := {δ(y, x) / y ∈ From, x ∈ X};   /* 5: compute the reached states.  */
     New := Y \ Reached;                 /* 6: set difference.              */
     if New = ∅ then return(True);       /* 7: test to empty set.           */
     Reached := Reached ∪ New;           /* 8: set union.                   */
     k := k + 1
  endloop;
end;
```

Figure 1. Breadth First Execution of the Product Machine for Comparison.

To compute the set of states `Y` and the generated ouputs `Z`, the enumeration of each input pattern of `X` and each state of `From` cannot be used for large machines because of the combinational explosion. In the same way, the set operations cannot be performed by set enumeration. In the following sections, we show how formal manipulations on boolean functions can solved these problems. Thus no enumeration will be performed. Moreover the underlying representation of boolean functions that we used is based on the Typed Decision Graphs which is the most efficient representation of boolean functions [3].

2.2. Symbolic Breadth First Execution

Since the two machines described by their LDS descriptions (the realization P_r and the specification P_S) are synchronous, the LDS description of the product machine is directly obtained by merging the two programs assuming that they have the same interface, and by defining the ouput function by the boolean function $Z =_{def} (Z_S \equiv Z_r)$ where Z_S and Z_r are the output functions of P_S and P_r.

In an LDS program the state encoding as well as the input and the output are *boolean vectors*, that is $\Sigma = \{True, False\}^n$, $O = \{True, False\}^m$, $S = \{True, False\}^p$ if the program has n inputs, m outputs and p state variables (that is LDS registers). In the same way, δ and λ are *vectorial boolean functions*, and they are obtained by *symbolic execution* of the program [4] [11]. So from the symbolic boolean vectors `From` and `X` we obtain the symbolic boolean vectors `Y` and `Z`. This means that we treat all the transitions and states reachable from the symbolic state `From` by using only one formal operation. In particular, no

explicit state or input enumeration is performed, and the symbolic execution replaces the statements 2, 3 and 5 of Figure 1. We describe in the next section how a boolean vector denotes a unique set and the formal manipulations on boolean functional vectors used to perform the set operations.

2.3. Set Manipulations: Converting a Functional Vector into a Set Characteristic Function

From here, we note $[f_1 \dots f_n]$ the vector whose k-th component is f_k, i denotes an interpretation, and I is the set of the interpretations. A boolean expression $e(y_1 \dots y_n)$ is a function from $\{True, False\}^n$ to $\{True, False\}$. It defines a subset $\chi(e)$ of $\{True, False\}^n$ by considering e as its *characteristic function*: $\chi(e) =_{def} \{i([y_1 \dots y_n]) / \models_i e(y_1, \dots, y_n)\}$. The manipulations on sets described by their characteristic functions are trivial. For example, we have $\chi(e_1) \cup \chi(e_2) = \chi(e_1 \vee e_2)$, $\chi(e_1) \setminus \chi(e_2) = \chi(e_1 \wedge \neg e_2)$, $\chi(e) = \emptyset$ iff $\models \neg e$, etc.

A functional boolean vector $F = [f_1 \dots f_n]$ is a function from I to $\{True, False\}^n$. We say that F defines a subset Set(F) of $\{True, False\}^n$ by $Set(F) =_{def} F(I)$. Until now, we do not know how to perform all the set operations directly on functional vectors, so we must convert them to the boolean expressions denoting the same set to treat the statements 4, 6, 7 and 8. Let $e(y_1, \dots, y_n)$ be the characteristic function of the set denoted by the functional vector F. We can assume that F uses the variables $\{x_1, \dots, x_m\}$ all different from $\{y_1, \dots, y_n\}$. The equation $Set(F) = \chi(e)$ is translated by the boolean equation:

$$\forall y_1 \dots \forall y_n, e(y_1, \dots, y_n) \Leftrightarrow (\exists x_1 \dots \exists x_m, (y_1 \equiv f_1) \wedge \dots \wedge (y_n \equiv f_n))$$

We use the equation solving facility on the TDG's [5] [12] to compute the function e: we perform an *existential abstraction* [5] of each variable x_k on the formula $\mathcal{F} =_{def} ((y_1 \equiv f_1) \wedge \dots \wedge (y_n \equiv f_n))$. Assuming that the variables $\{y_1, \dots, y_n\}$ are interpreted before those used by F, Figure 2a shows the TDG (noted *tree*) of $\mathcal{F}$.

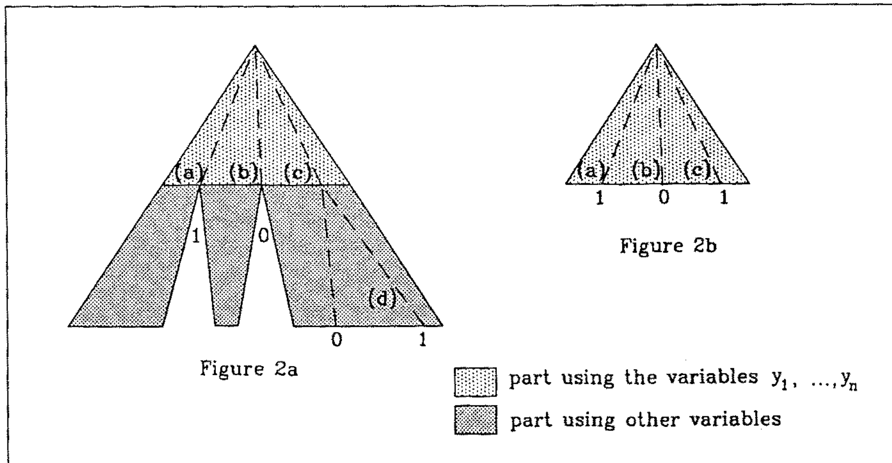

Figure 2. Computing the Characteristic Function of a Boolean Functional Vector.

The TDG of e given in Figure 2b is directly built from *tree* according to the type of each path starting from the root of *tree*:

- The path (a) reachs a leaf evaluated to True without meeting a variable used by F. This path defines a set of interpretations of $\{y_1, ..., y_n\}$ such that for each interpretation of $\{x_1, ..., x_m\}$, $\mathcal{F}$ is satisfied. Then for this path $e(y_1, ..., y_n)$ is satisfied, so we associate a leaf evaluated to True to (a).
- The path (b) reachs a leaf evaluated to False without meeting a variable used by F. This path defines a set of interpretations of $\{y_1, ..., y_n\}$ for which there is no interpretation of $\{x_1, ..., x_m\}$ satisfying $\mathcal{F}$. Then for this path $e(y_1, ..., y_n)$ does not hold, so a leaf evaluated to False is associated to (b).
- The path (c) meets a variable used by F. Because of the canonicity of the TDG's, there exists at least one subpath (d) in the subtree starting from this variable that reachs a leaf evaluated to True. It results that for the set of interpretations of $\{y_1, ..., y_n\}$ defined by (c), the path (d) provides some interpretation of $\{x_1, ..., x_m\}$ satisfying $\mathcal{F}$. Then for the path (c) $e(y_1, ..., y_n)$ is satisfied, so the subtree is replaced by a leaf evaluated to True.

For example, consider the functional vector $F = [(x_1 \vee x_2)\ (x_1 \oplus x_2)]$. We have Set(F) = {[True False], [True True], [False False]}, and its characteristic function $e = (y_1 \vee \neg y_2)$ is obtained directly by using the algorithm described above.

2.4. Simplifying a boolean function under a constraint

As previously stated, it is desirable to find the most fitted set of states `From` defined by "**choose** `From` **in such a way that** `New` ⊂ `From` ⊂ `Reached`" in the general algorithm of Figure 1. In this algorithm, `New` and `Reached` are boolean expressions, that is characteristic functions. On the other hand, λ and δ are vectorial boolean functions and `X`, `From`, `Z` and `Y` are functional vectors. Our aim is to find a functional vector `From` respecting the condition χ(`New`) ⊂ Set(`From`) ⊂ χ(`Reached`) and having a functional complexity as low as possible, that is the most compact possible tree representation.

We consider here the general problem of simplifying a function f under a constraint c. This means that we want to find a function noted f/c such that f and f/c are equal on the domain defined by the constraint c, that is (E) $\models c \Rightarrow (f/c \equiv f)$, and the TDG of f/c has a size less than or equal to that of f. We assume that $c \neq$ False, because there is no sense to define a function on an empty domain. The process of restriction can be seen as a *partial evaluation* of f under the constraint c in the following way. If c = True, the domain where $f/c \equiv f$ is that of f, so f/c = f. If f is the constant function True or False, then f/c is the same constant function. If f and c are not constant functions, then there exists a variable x which occurs in f or c. We note $(\neg x \wedge f_0) \vee (x \wedge f_1)$ and $(\neg x \wedge L) \vee (x \wedge H)$ the respective Shannon's Canonical Form of f and c. Then the three following cases enable us to recursively compute f/c:

- if x occurs in f and not in c (that is $f_0 \neq f_1$ and L = H), then $f/c = (\neg x \wedge f_0/c) \vee (x \wedge f_1/c)$ because (E) must be respected whatever the interpretation of x.
- if x occurs in c and not in f (that is $f_0 = f_1$ and $L \neq H$), then f does not depend on the interpretation of x, and (E) permits us to write that $f/c = f/(L \vee H)$.

- Else x occurs in f and c (that is $f_0 \neq f_1$ and $L \neq H$). If L = False then $f/c = f_1/H$ because the interpretations where x = False do not satisfy c. In the same way, if H = False then $f/c = f_0/L$. These two cases reduce the size of f/c. Otherwise, $f/c = (\neg x \wedge f_0/L) \vee (x \wedge f_1/H)$.

It is easy to show inductively that this definition of f/c makes (E) hold and that the size of the TDG of f/c is less than or equal to that of f. More precisely, f can be simplified on the domain where c is not satisfied. For example, the function $f = (x_2 \wedge (x_1 \Leftrightarrow (x_3 \Rightarrow x_4))) \vee \neg (x_2 \vee ((x_4 \Rightarrow x_1) \wedge (x_1 \vee x_3)))$ restricted under the constraint $c = (\neg x_2 \wedge x_3 \wedge x_4) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$ is the function $f/c = x_1 \wedge x_2$. We can notice that f/c is different from $f \wedge c$, and that its TDG is more compact than that of $f \wedge c$.

2.5. The Final Proof Algorithm

In the general proof algorithm, the sets of states Y and From are functional vectors defined over a subset of the input variables previously introduced and so, if not controlled, their functional complexity may be important. To avoid such an explosion, we must choose From of lowest complexity respecting the equation $\chi(\text{New}) \subset \text{Set}(\text{From}) \subset \chi(\text{Reached})$. Such an adequate functional vector can be obtained by simplifying $Y = [f_1 \ldots f_n]$ under a constraint translating that we must have $\chi(\text{New}) \subset \text{Set}(Y) \subset \chi(\text{Reached})$. We have shown that a boolean function can be simplified on the domain where its constraint is not satisfied, so we must choose the most restrictive constraint, that is the boolean expression New. In order to have a constraint using the variables of Y (if not, no simplification is possible), we convert the constraint New by substituting in $\text{New}(y_1, \ldots, y_n)$ each occurence of y_k by its corresponding boolean function f_k, that is we evaluate $\text{New}(f_1, \ldots, f_n)$. This operation treats the statement 1 of Figure 1.

```
/* Functional boolean vectors are underlined, characteristic functions are in  */
/* italic. The functions AndTDG, NotTDG, OrTDG apply logic connectors on TDG's */
function formal-prove(M : Finite-State-Machine) : boolean;
var k : int;
    Reached, New : TDG;
    X, Y, Z, From : array of TDG;
begin
  k := 0;  From := M.init;  Reached := get-charac(From);
  do loop
      X := new-inputs(k);          /* generate a new symbolic vector.        */
      Z := λ(From, X);             /* compute the symbolic output vector.    */
      if Z ≠ [True ... True] then return(False);
      Y := δ(From, X);             /* compute the symbolic state vector.     */
      New := AndTDG(get-charac(Y), NotTDG(Reached));   /* set difference.    */
      if New = False then return(True);                /* test to empty set. */
      Reached := OrTDG(Reached, New);                  /* set union.         */
      From := restrict-vector(Y, evaluate(New,Y));
      k := k + 1
  endloop;
end;
```

Figure 3. The Final Proof Algorithm Using Symbolic Manipulations.

The final version of our proof algorithm using only formal manipulations on boolean functions is given in Figure 3. Our choice of From reuses the input variables already introduced in the program to

optimize the sharing in memory of all the manipulated TDG's. The function get-charac(F) returns the characteristic function associated to a functional vector F as described in Section 2.3. The function evaluate($e(y_1, ..., y_n)$, [f_1 ... f_n]) returns $e(f_1, ..., f_n)$, and restrict-vector([f_1 ... f_n], c) returns the vector [f_1/c ... f_n/c].

3. Example and Result

As a simple example, we consider a BCD recognizer. The LDS description of an implementation [7] is described in Figure 4. It has one input X and one output Z. The output Z is equal to 1 until the reading of a sequence of four inputs is validated. At this moment, Z is equal to 1 if and only if the current input and the three previous inputs form a BCD number.

```
Component BCDR;
Interface;             ? Input and output ports
  X, IN;  Z, OUT;
end;
Registers;             ? Storage elements of the circuit
  YR1, YR2, YR3, BIN;  ? BIN denotes binary values
end;
Variables;             ? Algorithmic variables
  Y01, Y02, Y03, Y04, Y05, Y06, Y07, Y08, Y11, Y12,
  Y13, Y14, Y15, Y21, Y22, Y23, Y31, Y32, Y41, BIN;
end;
Init;                  ? Initial state of the circuit
  YR1 := 0; YR2 := 0; YR3 := 0;
end;
        ? The logical NOT operator is #, AND is .,OR is /
begin
  Y01 := # X; Y02 := # X; Y03 := # YR3; Y04 := # YR1; Y05 := # YR2; Y06 := # YR3;
  Y07 := # YR1; Y08 := # YR3; Y11 := Y01 . YR1 . YR3; Y12 := Y02 . YR2 . Y03;
  Y13 := Y04 . Y05 . Y06; Y14 := X . Y07 . Y08; Y15 := X . YR1 . YR3;
  Y21 := Y11 / Y12; Y22 := Y13 / Y14; Y23 := YR2 / Y15; YR1 := Y21; YR2 := Y22;
  YR3 := Y23; Y31 := # YR1; Y32 := # YR2; Y41 := X . Y31 . Y32 . YR3; Z := # Y41;
end;
```

Figure 4. LDS Description of the Realization of the BCD Recognizer.

An explicit specification BCDS of such a circuit can be written as in Figure 5. We can notice that this machine has several starting states since Init does not constraint the registers Y, so Y is initially symbolic. Clearly it does not exist direct relation between the state of the specification and the register encoding used by the realization. The specification is decomposed into different modules whereas the realization is a flat description.

The product machine uses the state variables of the two programs, that is YR1, YR2, YR3, compteur and Y. From the symbolic initial product state the product machine is symbolically executed and the outputs of each machine are compared for equivalence at each step until no new product state is reached. The equivalence proof is performed in 0.5 s.

```
Component BCDS;
Interface;                        ? The inputs and outputs of the program
  X, IN;  Z, OUT;
end;
Registers;                        ? Storage variables
  compteur 0:2, BIN; Y 0:3, BIN;  ? BIN denotes binary values
end;
Init;                 ? Initial state. Y is not specified
  compteur 0:2 := '00';
end;

Function INCR (VAR X 0:2, BIN);   ? Var denotes variable parameter
begin
  X := X + 1;
end;

Function BCD (X 0:4, BIN);        ? Test if X is a BCD number
begin
  return(# X 0 / # X 1 . # X 2);
end;

Function SHIFT (X, BIN; VAR Y 0:3, BIN)
begin
  Y 2 := Y 1; Y 1 := Y 0; Y 0 := X;
end;

begin                              ? Body of BCDS
  Z := (compteur <> 3) / BCD(X ! Y); ? concatenation of fields is !
  Call SHIFT(X,Y);
  Call INCR(compteur);
end;
```

Figure 5. Specification of the BCD Recognizer

The use of formal treatments in our method enables us to check various properties. For example, we can verify that there is only one starting state (modulo the state of Y) of the two machines such that they are equivalent. We can verify at each step that the outputs satisfy a condition translated by a boolean expression. We can constraint the inputs by computing their partial boolean functions satisfying some properties which can describe for example the behaviour of the environment of the circuit.

This proof algorithm has compared the specification and the realization of a a memory controller (10 inputs, 15 registers, 5 outputs) in 30 minutes. It requires 11 symbolic cycles, that means 110 input variables have been introduced to reach all the useful states of the product machine for any input sequence. The comparison has detected a mistreatment of errors, and a dependancy of certain states to one input while the specification did not apply any constraint on it.

Of course, the Symbolic Breath First Execution is very sensitive to the connexity of the state diagram of the machine. If the state diagram is strongly connected, then the symbolic breath first execution of the machine is very efficient, because formal manipulations treat many states and transitions at each step. In the worst case of our technique, the machine reachs only one state at each step until it

comes back to its initial state (such as a pulse counter). For example, the traversal of a modulo 256 pulse counter requires 30 s.

Conclusion

In this paper, we have presented a method for comparing two sequential machines using a *symbolic execution* of the product machine. The symbolic execution is supported by *formal manipulations* on boolean expressions and boolean functional vectors that avoid enumeration methods. The algorithms used in these formal manipulations come from equation solving [5], and we are now studying how they can be made more efficient.

References

[1] F. Anceau, "Design Methodology for Large Custom Processors". *Proc. of the 1986 ESSIR Conference*, Delft.

[2] J. R. Armstrong, *Chip-Level Modeling with VHDL*, Prentice Hall, 1989.

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

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

[5] O. Coudert, J. C. Madre, "Logics over Finite Domain of Interpretation: Proof and Resolution Procedures", *BULL Research Report* to appear, 1989.

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

[7] D. Dietmeyer, *Logic Design of Digital Systems*, Allyn & Bacon, 2nd edition, 1978.

[8] J. E. Hopcroft, J. D. Ullman, *Introduction to Automata Theory, Langages and Computation*, Addison-Wesley, Reading, Mass., 1979.

[9] S. H. Hwang, A. R. Newton, "An efficient Design Correctness Checker of Finite State Machines", ICCAD 1987.

[10] D. Jaillet, P. Mertens, *LDS Reference Manual*, BULL S.A., May 1987.

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

[12] J. C. Madre, O. Coudert, "Formal Verification of Digital Circuits Using a Propositional Theorem Prover", *IFIP Working Conference on the CAD Systems Using AI Techniques*, June 1989.

[13] J. Y. Murzin, "FAON A functional Abstractor of Netlist", *Actes du Séminaire de Programmation Logique*, Lannion, France, 1986.

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