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, a new set of formal computation on boolean functions represented by Typed Decision Graphs has been defined.

1. Introduction

It is now commonly admitted that 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 by the designers. 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. In order to go beyond this limit, BULL is making an important effort in research and development of formal methods for the verification of its hardware designs.

Formal verification of hardware consists in comparing (for equivalence or implication) 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]. This language was designed to describe the structure and the behaviour of synchronous digital circuits at the Register Transfer Level. 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. An LDS program is very similar to a VHDL block statement [2]. LDS programs can be compiled and executed on input patterns provided by the circuit designer. In the recent few years, several tools have been developed at BULL to extract behavioral descriptions from structural (at layout, transistor or gate levels) descriptions [14]. This extraction process produces an LDS program which translates exactly the functional behavior of the realized circuit.
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.

The specification of a circuit reflects the intentions of the designers. It is a more abstracted description than the one extracted from the structure. The specification has the same interface (inputs and outputs) as the realization. It is usually shorter because it uses more complex data structures and statements. For instance, a specification may use arithmetic operators, complex data structures (enumerated types, indexed sets, records ...) and iterative statements, whereas the extracted program only uses bit vectors, logical operators and possibly conditional statements. The specification may also use functions and procedures; in a similar way to structured programming in software development, the specification of an hardware device using an Hardware Description Language such as LDS is easier to give when described as a hierarchy of communicating procedures.

In some cases, the specification of the circuit behavior may be partially defined: the machine is "incompletely specified". This is due to the particular environment the circuit is intended to work with. Taking into account these operating conditions is generally mandatory to prove that a circuit is correct with respect to its specification.

Until now, the approach to formal verification taken by BULL was limited to combinational circuits. A powerful prover called PRIAM has been developed at BULL's Research Center [3] [12]. PRIAM can handle industrial circuits such as 64 bits ALU's and other data-path operators with up to 20000 transistors. The core of PRIAM is a powerful theorem prover in propositional logic. Boolean expressions are represented by Typed Decision Graphs (TDG) [3] [4], the most compact representation currently known.

PRIAM has a severe limitation: it is able to compare a specification program with the extracted one only if both programs have the same registers and the same state encoding. This requirement is very difficult to satisfy: it implies a perfect knowledge of the machine encoding. First, the specification generally is not precise enough; it does not go down to the actual encoding. Rather, the specification represents some form of high level state diagrams. Second, the realization and the specification may not have the same hierarchical structure. Even though the design process generally follows the initial specification, the various optimizations performed during the design process may modify significantly the hierarchy of the specification. When the design of the circuit is completed, the specified hierarchy does not necessarily reflect the circuit decomposition, in particular, the state variables generally do not coincide.

Thus, 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

First, we recall the usual definitions of sequential machines [8] [11]. A deterministic finite state machine \( M \) is defined by the 6-tuple \((\Sigma, O, S, s_0, \delta, \lambda)\) where \( \Sigma \) is the input alphabet, \( O \) the output alphabet, \( S \) the finite set of states of the machine, \( s_0 \) the initial state, \( \delta \) the transition function from \( S \times \Sigma \) to \( S \), and \( \lambda \) the output function from \( S \times \Sigma \) to \( O \). The functions \( \delta \) and \( \lambda \) can be partial if some transitions or outputs of the machine are not specified. We say that the machine \( M \) accepts the input sequence \( x_1 \ldots x_n \) if and only if there exists a sequence of states rooted at the initial state \( s_0 \ldots s_n \) such that \( 1 \leq k \leq n, s_k = \delta(s_{k-1}, x_k) \) is well defined.

When an input sequence \( x_1 \ldots x_n \) is accepted by \( M \), the corresponding output sequence \( z_1 \ldots z_n \) generated by \( M \) when reading the input sequence is such that for \( 1 \leq k \leq n, z_k = \lambda(s_{k-1}, x_k) \). If \( \lambda \) is a partial function, this output sequence can be undefined. We introduce the extra symbol \( \bot \) not belonging to \( O \) to denote the undeterminate value of \( \lambda \). We say that \( z_1 \) is included in \( z_2 \) iff \( z_1 = z_2 \) or \( z_1 = \bot \).
For two machines $M_1$ and $M_2$ operating on the same input and output alphabets, we say that $M_1$ implies $M_2$ iff each input sequence $x_1...x_n$ in $\Sigma^*$ accepted by $M_1$ is accepted by $M_2$ and the output sequences $z_1...z_m$ and $z_2...z_n$ of $O^*$ are such that $z_{1k}$ is included in $z_{2k}$. Most of the time, the verification of a sequential machine consists in proving that its realization is implied by its specification. When a total equivalence between a specification $M_1$ and a realization $M_2$ is required, it must be shown that $M_1$ implies $M_2$ and $M_2$ implies $M_1$.

The usual method for proving the equivalence between two finite state machines $M_1 = (\Sigma, O, \delta_1, \lambda_1, S_1, s_0)$ and $M_2 = (\Sigma, O, \delta_2, \lambda_2, S_2, s_0')$ consists in building the product machine $M = (\Sigma, O \times O, \delta, \lambda, S, s_0)$ where the set of states $S = S_1 \times S_2$, $s_0 = (s_0, s_0')$, the output function $\lambda((s_1, s_2), x) = (\lambda_1(s_1, x), \lambda_2(s_2, x))$ and the transition function $\delta(((s_1, s_2), x) = (\delta_1(s_1, x), \delta_2(s_2, x))$. Then the machines $M_1$ and $M_2$ are equivalent if and only if for every transition of $M$ that can be reached from the initial product state $(s_0, s_0')$, the machine $M$ produces an output $(z, z)$. 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 [15]. Several methods have been proposed to enumerate the states of this machine without building it. There are mainly two enumeration procedures, the depth first and the breadth first enumerations. 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 first present the breadth first traversal of a sequential machine. 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. This algorithm uses only set operations. It considers all the states and transitions reachable from the set $M$.init of initial states. At each step, the set of reachable states $Y$ from the set $\text{From}$ is computed. The set $\text{New}$ of newly discovered states is also computed. Then the set $\text{Reached}$ is enlarged by $\text{New}$.

```
function prove(M : Finite-State-Machine) : boolean;

var k : int;
From, Reached, X, New : Set-Of-States;
Z : Set-Of-Outputs;
X : Set-Of-Inputs;

begin
k := 0; New := M.init; Reached := New;
do loop
  From := New;
  X := new-inputs(k);
  Z := (\lambda(y,x) / y \in From, x \in X);
  if not correct(Z) then return(False);
  Y := \delta(y,x) / y \in From, x \in X;
  New := Y \setminus Reached;
  if New = \emptyset then return(True);
  Reached := Reached \cup New;
  k := k + 1
endloop;
end;
```

Figure 1. Breadth First Execution of the Product Machine.
Clearly each reachable product state is considered, so the proof of equivalence (or implication) is complete. The execution process stops either when the two machines generate different outputs or when no new product state is reached. In that case both machines are equivalent (or implying), 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 set of states \( \text{From} \) with which the next set of reached states \( \text{New} \) is computed must satisfy the following property: it must contain the set \( \text{New} \) (newly reached states) and may contain any already discovered state belonging to \( \text{Reached} \). So the proof remains correct and complete if the statement \( "\text{From} := \text{New}" \) is replaced by "choose \( \text{From} \) in such a way that \( \text{New} \subseteq \text{From} \subseteq \text{Reached} \)."

This algorithm must enumerate each input pattern of \( x \) and each state of \( \text{From} \) to compute the set \( x \) and the set of generated outputs \( z \). Enumeration methods cannot be used for large machines because of the combinatorial complexity. The method explained in the next section essentially uses this algorithm \( \text{but treats all the sets as boolean expressions or boolean functional vectors} \). The set operations are performed through symbolic manipulations. Thus no enumeration is 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 obtained by merging the two programs assuming that they have the same interface, and by defining the output function by the boolean function \( Z = \text{def} (Z_s \equiv Z_t) \) for equivalence and \( Z = \text{def} (Z_s \land Z_t) \) for implication, where \( Z_s \) and \( Z_t \) are the output functions of \( P_s \) and \( P_t \). Thus the problem of checking the equivalence or implication of two machines comes down to verify that this product machine produces an output equal to True for any transition and state reachable from its set of initial states.

In an LDS program the state encoding \( x \) as well as the input \( x \) and output \( z \) are \( \text{boolean vectors} \), that is \( \Sigma = (\text{True}, \text{False})^n \), \( O = (\text{True}, \text{False})^m \), \( S = (\text{True}, \text{False})^p \) if the program has \( n \) inputs, \( m \) outputs and \( p \) state variables (that is LDS registers). In the same way, \( \delta \) and \( \lambda \) are \( \text{vectorial boolean functions} \). A LDS program describes how the partial transitions function \( \delta \) and the partial output function \( \lambda \) can be computed from its inputs and its state variables. Our method uses directly the vectorial functions \( \delta \) and \( \lambda \) produced by the symbolic execution of the program [3] [5] to compute the sets \( x \) and \( z \) from the sets \( x \) and \( \text{From} \), where these sets are denoted by \( \text{boolean functional vectors} \). This means that we treat all the transitions and states reachable from the set \( \text{From} \) by using only one formal operation. In particular, no explicit state enumeration is performed. We describe in the next section the \( \text{formal manipulations on boolean functional vectors} \) used to \( \text{symbolically execute} \) the product machine.

2.2.1 Converting a Functional Vector into a Set Characteristic Function

From here, we note \([x_1 \ldots x_m] \) the vector whose \( k \)-th component is \( x_k \). A boolean expression \( e \) using the formal variables \( y_1 \ldots y_n \) is a function from \( (\text{True}, \text{False})^n \) to \( (\text{True}, \text{False}) \). It defines a subset \( \chi(e) \in (\text{True}, \text{False})^n \) by considering \( e \) as its characteristic function:

\[
\chi(e) = \text{def} \{ i \in [y_1 \ldots y_n] \mid t(e(y_1, \ldots, y_n)) \}
\]

The manipulations on sets described by their characteristic functions are trivial. For example, we have \( \chi(e_1) \cup \chi(e_2) = \chi(e_1 \lor e_2) \), \( \chi(e_1) \setminus \chi(e_2) = \chi(e_1 \land \neg e_2) \), \( \chi(e) = \emptyset \) iff \( \models \neg e \), etc.

A functional boolean vector \( F = [f_1 \ldots f_n] \) is a function from \( \mathcal{V} \times I \) to \( (\text{True}, \text{False})^n \), where \( \mathcal{V} \) is the set of propositional symbols and \( I \) the set of valuations, that is the set of functions from \( \mathcal{V} \) to \( (\text{True}, \text{False}) \). We say that \( F \) defines a subset of \( (\text{True}, \text{False})^n \) by:
Set(F) = \text{def } F(\mathcal{P} \times I)

An efficient way to manipulate sets described by functional vectors is to convert a functional vector to the boolean expression denoting the same set. Let \( \varphi \) be the function which associates the unique (modulo the choice of its variables) boolean expression \( e \) to the functional boolean vector \( F \) such as \( \text{Set}(F) = \chi(e) \). Of course \( \varphi \) is not injective. Let us choose \( n \) variables \( y_1, ..., y_n \) of \( e \) such that they are different from the variables occurring in \( F \) which are denoted by \( \text{Var}(F) \). We note \( i \) an interpretation of the variables \( y_1, ..., y_n \) that is a function from \( \{y_1, ..., y_n\} \) to \{True, False\}, and \( j \) is an interpretation of \( \mathcal{P} \setminus \{y_1, ..., y_n\} \) that is a function from \( \mathcal{P} \setminus \{y_1, ..., y_n\} \) to \{True, False\}. In particular we have \( \text{Var}(F) \subset \mathcal{P} \setminus \{y_1, ..., y_n\} \). The function \( i \cup j \) defined by \( (i \cup j)(x) = i(x) \) if \( x \in \{y_1, ..., y_n\} \) and \( (i \cup j)(x) = j(x) \) if \( x \notin \{y_1, ..., y_n\} \) is an interpretation of \( \mathcal{P} \). Then the equation \( \text{Set}(F) = \chi(e) \) defines the function \( \varphi \) by the following equation:

\[ \models \varphi(F) \iff \exists j, \models i \cup j(y_1 \equiv f_1) \land ... \land (y_n \equiv f_n) \]

So the computation of \( j(F) \) comes down to a problem of satisfiability.

The canonicity of our representation is exploited to solve efficiently this equation [13]. Assuming that the variables \( \{y_1, ..., y_n\} \) are interpreted before those used by \( F \), Figure 2a shows the form of the Shannon's canonical tree [3] (noted treeF) of the formula \( \mathcal{F} = \text{def } (y_1 \equiv f_1) \land ... \land (y_n \equiv f_n) \). The formula \( \varphi(F) \) is directly built from treeF. Each path from the root of treeF has one of the three following properties as described in Figure 2a and the Shannon's canonical tree of \( \varphi(F) \) is given in Figure 2b.

\[\begin{array}{c}
\text{Figure 2a} \\
(0) (1) (c) \\
1 0 1
\end{array}\]

\[\begin{array}{c}
\text{Figure 2b} \\
(a) (1) (c) \\
0 1
\end{array}\]

- The path (a) reaches a leaf evaluated to True without meeting a variable of \( F \). This path defines a set of interpretations \( i \) such that for each interpretation \( j \), \( F \) is satisfied. So we associate a leaf evaluated to True to this path.

- The path (b) reaches a leaf evaluated to False without meeting a variable of \( F \). This path defines a set of interpretations \( i \) for which there is no interpretation \( j \) satisfying \( F \). So we associate a leaf evaluated to False to this path.
- The path (c) meets a variable used by F. Because of the canonicity of the representation, there exists at least one subpath (d) in the subtree starting from this node which reaches a leaf evaluated to True. Thus, for the set of interpretations j defined by the path, there exists at least one interpretation j satisfying j, so the subtree is replaced by a leaf evaluated to True.

Figure 3 shows the function cut which computes \( \varphi(F) \) from \( \text{tree}_F \) in Shannon's canonical form. This algorithm cuts the branches involving variables of F (that is the variables whose order is greater than or equal to \( \text{cut-order} \)) and replaces them by leaves evaluated to True. Of course, our representation of boolean functions by TDGs is much more efficient, and the algorithm evaluating \( \varphi \) uses a method which shares the computing of subtrees and frees them as soon as possible to optimize the memory management and reduce the time of computing.

```
var cut-order : int;

function cut(tree : vertex) : vertex;
begin
  if tree = True then return(True);
  if tree = False then return(False);
  if order(tree.root) >= cut-order then return(True);
  return(disj(conj(not(tree.root), cut(tree.low))
               conj(tree.root, cut(tree.high))));
end;
```

Figure 3. Algorithm of Abstraction of some Variables in a Boolean Expression.

For example, consider the functional vector \( F = [(x_1 \lor x_2) (x_1 \oplus x_2)] \). We have Set(F) = \([T,F],[T,T],[F,F]\). In order to compute the characteristic boolean expression \( \varphi(F) \) of this set, we introduce the variables \( y_1 \) and \( y_2 \). Figure 4a gives the Shannon's canonical representation of the formula \( F = [y_1 \land y_2] \). Figure 4b shows the Shannon's canonical tree of \( \varphi(F) \) (that is \( \varphi(F) = y_1 \lor \neg y_2 \)) obtained directly from the previous one by using the cut algorithm described above.

```
  y1
   \--- y2
    \-- 0
     \-- x1
      \-- 0
       \-- x2
        \-- 1
         \-- 0
          \-- 1
           \-- 0
            \-- 1
             \-- 1
              \-- 0

(b)
```

Figure 4. Abstraction of the Satisfiability on the Variables \( x_k \)

### 2.2.2 Simplifying a boolean function under a constraint

As previously stated, it is desirable to find the most fitted set of states \( \text{From} \) defined by "choose \( \text{From} \) in such a way that \( \text{New} \subset \text{From} \subset \text{Reached} \)" in the general algorithm of Figure 1. In this algorithm, \( \text{New} \) and \( \text{Reached} \) are boolean expressions denoting characteristic functions. On the other hand, \( \lambda \) and \( \delta \) are vectorial boolean functions, so \( X, \text{From}, Z \) and \( Y \) are
functional vectors. We know how to apply set operators on sets described by functional vectors by converting them into their characteristic functions. Our aim is to find a functional vector \( f \) from respecting the condition \( \chi(\text{New}) \subseteq \text{Set}(f) \subseteq \chi(\text{Reached}) \) and having a functional complexity as low as possible. The functional complexity criterion consists in obtaining a functional vector \( f \) from with 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:

- The functions \( f \) and \( f/c \) are equal on the domain defined by the constraint \( c \), that is:

\[
(\text{E1}) \quad f(c) = (f/c)(c).
\]

- The representation of \( f/c \) has a complexity less than or equal to that of \( f \). In other words, the size of the representation of \( f/c \) is in the worst case equal to that of \( f \).

The tautology (E1) which must be respected can be written as:

\[
(\text{E2}) \forall i, i \models c \Rightarrow i(f/c) = f
\]

where \( i \) is any interpretation. So the restriction of the representation of \( f \) can be made only with the interpretation satisfying \( \neg c \).

The process of restriction can be seen as a partial evaluation of \( f \) under the constraint \( c \) in the following way.

- If \( c = \text{False} \), the domain where \( f/c \) is defined is empty and there is no sense to define a function on an empty domain.

- If \( c = \text{True} \), the domain where \( f/c = f \) 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 have the three following cases which enable us to recursively compute \( f/c \):

  - If \( x \) occurs in \( f \) and \( c \), then the Shannon’s canonical form of \( f \) and \( c \) can be respectively written as \( (\neg x \land f_0) \lor (x \land f_1) \) and \( (\neg x \land L) \lor (x \land H) \). If \( L = \text{False} \) then \( f/c = f_1/H \) because the interpretations where \( x = \text{False} \) do not satisfy \( c \). If \( H = \text{False} \) then \( f/c = f_0/L \) because the interpretations where \( x = \text{True} \) do not satisfy \( c \). These two cases reduce the size of \( f/c \) with respect of \( f \) and satisfy (E2). Otherwise, \( f/c = (\neg x \land f_0/L) \lor (x \land f_1/H) \).

  - If \( x \) occurs if \( f \) and not in \( c \), then (E2) must be respected whatever the interpretation of \( x \), so \( f/c = (\neg x \land f_0/c) \lor (x \land f_1/c) \).

  - If \( x \) occurs in \( c \) and not in \( f \), then \( f \) does not depend on the interpretation of \( x \), and (E2) permits us to write that \( f/c = f(L \lor H) \).

It is easy to show inductively that this definition of \( f/c \) makes (E1) hold and that the size of the representation of \( f/c \) is less than or equal to that of \( f \). Figure 5 shows an example of a restriction using Shannon’s Canonical Forms. We have \( c = (\neg x_2 \land x_3 \land x_4) \lor (x_2 \land (x_3 \equiv x_4)) \) and \( f = x_2 \land (x_1 \equiv (x_3 \Rightarrow x_4)) \lor (x_2 \lor ((x_4 \Rightarrow x_1) \land (x_1 \lor x_5))) \). The obtained restricted function \( f/c \) is \( x_1 \land x_2 \). We can notice that \( f/c \) is different from \( f \land c \), and that its representation is more efficient than that of \( f \land c \).