An automatic generalization method for the inductive proof of replicated and parallel architectures

Laurence PIERRE

Laboratoire d'Informatique de Marseille - URA CNRS 1787
CMI / Université de Provence
Technopôle de Château-Gombert, 39 rue Joliot-Curie
13453 Marseille Cedex 13 - FRANCE
e-mail : laurence@gyptis.univ-mrs.fr

Abstract : Our approach for verifying the equivalence of two VHDL architectures consists in translating these descriptions into functional forms and then in proving the equivalence of these functions. As far as replicated or parallel architectures are concerned, an induction-based method is used for verifying generic n-bit descriptions. This technique takes advantage of the regular structure of these devices and can give better results than the BDD-based approaches. However, induction requires complete specifications, whereas the designers usually supply partial specifications. Therefore, we propose a specialized automatic method for generalizing such incomplete statements, before the Boyer-Moore proof process.

I. INTRODUCTION

The concept of formal verification includes various aspects [9], this paper focuses on verifying that the function implemented by a given VLSI circuit implies or is equivalent to its specification. Our system PREVAIL™, that is being developed in cooperation with Imag/Artemis, is a prototype proof environment which includes a set of verification tools [5]. It takes as input VHDL descriptions [13] and verifies the equivalence (or the implication) of two different *architectures* of the same *entity* (two descriptions given at different abstraction levels, or a description and its optimization, etc...). The most appropriate verification tool is selected with the help of the designer, and the descriptions are automatically translated into the formalism of this proof system. One of the included proof tools is the Boyer-Moore system, Nqthm [3][4].

This system, which is essentially based on the induction principle, is particularly useful for the verification of generic replicated or parallel architectures. In effect, such a theorem prover bring several advantages :

- we do not have to care about the size of the device, we recursively describe generic n-bit architectures and we prove generic properties,
- the specification can be given at the arithmetic level, and we can use conversion functions between bit-vectors and integers.
  This feature is very important for the hierarchical verification of large arithmetic devices : after being verified, each elementary component can be replaced by its arithmetic specification. Thus, the whole system verification is performed at the arithmetic level, and the efficiency is significantly improved.

This approach circumvents some drawbacks of the well-known tautology-checking techniques based on BDD-like representations [6] (see for instance [10], [17]), where :

- the size of the circuit must be fixed to 16, 32, 64... bits, and the proof system verifies the correctness of each output independently,
- the "specification" of the circuit must be another implementation of the same arithmetical function, also considered at the boolean level. Usually, even if the specification is given at the arithmetical level, it has to be transformed into an equivalent bit-level implementation, using a library of simple basic modules (adders, multipliers,...).

Nevertheless, a correct recursive model cannot be deduced from a simple netlist of the circuit, and our approach requires a particular description methodology. The automatic translation task is feasible provided that some description rules are respected. In particular, the genericity of the architecture must be expressed by a declaration of the form `generic(N:positive)`, and the regularity must be described using a `for ... generate` construct. Moreover, the theorems to be verified are usually expressed in terms of given values for the incoming boolean signals (input carries). It means that induction cannot be applied, unless we are able to provide a generalized version where carries are considered as symbolic variables. Generalization is a well-known problem in the framework of induction-based proofs, and no general-purpose method can be found. Conversely, it may be possible to mechanize some specialized methods. Here, we propose an automatic technique for generalizing the partial specifications of one-dimensional, as well as two-dimensional, replicated structures. This method extends a previous work on simple one-dimensional regular devices [19].

Finally, our methodology is able to give a positive answer to one of the questions that have been asked in the presentation of the N-bit adder TPCD benchmark [14] : "Is the verification done for arbitrary N?". We will use this simple ripple-carry adder as running example throughout this paper; its VHDL description is given by Figure 1. More elaborate examples can be found in paragraph V.

```
entity rca is
  generic(N:positive);
  port(a,b:in bit_vector(0 to N-1); c:in bit;
       z:out bit_vector(0 to N));
end rca;

architecture struct of rca is
component fulladd
  port(a,b,carry:in bit; fout,fcarry:out bit);
end component;
signal carry:bit_vector(0 to N);
begin
   addblock:block
   for all:fulladd use entity work.fulladd(stadd);
   begin      --- Regular part :
     fadd:for I in 0 to N-1  generate
        f1:fulladd port map(a(I),b(I),carry(I),z(I),carry(I+1));
     end generate;
     carry(0) <= c;
     z(N) <= carry(N);
   end block;
end struct;
```

FIGURE 1 : VHDL DESCRIPTION OF THE RIPPLE-CARRY ADDER

II. BRIEF OVERVIEW OF THE BOYER-MOORE SYSTEM

All along this paper, we only have to suppose that the proofs are performed in an induction-based environment. In fact, we use Nqthm and the reader will find some lines of Boyer-Moore code. Therefore, we give a brief overview of this system.

The Boyer-Moore theorem prover is based on a quantifier-free first order logic with equality. The three basic principles of this prover are [3][4] :

- **The "shell" principle** :
  Inductive abstract data types - called "shells" - can be built by means of a bottom object, a constructor, and one or more accessors. A boolean function, called a recognizer, recognizes if an object belongs to the shell.
  The type of "natural numbers" is a well-known example : the bottom is 0, the constructor is $\lambda\, x.\ x{+}1$, and the accessor is $\lambda\, x.\ x{-}1$; the predicate recognizer of this type in Boyer-Moore is *numberp*.
- **The definition principle** :
  Prior to accepting a recursive function definition, the system verifies that there exists a measure which decreases according to a "well-founded" relation.
  For example, we can define the following function "times" over natural numbers :
  $$times\ (i, j)\ =_{def}\ if\ i = 0\ then\ 0\ else\ j + times\ (i\text{-}1, j)$$
  The term i-1 decreases at each recursive call, and the recursion stops when i=0. Thus, this recursive definition is correct. Much more elaborate function templates can also be accepted.
- **The induction principle** :
  It allows to prove inductive theorems over recursive functions. Induction schemes are *automatically* generated according to the definitions of the recursive functions involved in the theorems. For example, let us assume that we want to verify the following proposition P(x,y) :
  $$numberp(x)\ and\ numberp(y)\ \Rightarrow\ times(x, y{+}1) = x + times(x, y)$$
  The induction scheme generated for the proof of P(x,y) is :
  1. $x = 0\ \Rightarrow\ P(x,y)$
  2. $(x \neq 0)$ and $P(x\text{-}1,y)\ \Rightarrow\ P(x,y)$

A brief state of the art about Nqthm and the formal proof of software/hardware can be found in [20].

Remark : In the following, we will have to use bit-vectors. The bit-vector "shell" has been proposed in [11]. It is defined by means of : the bottom (**btm**), the constructor **bitv**, the accessors **bit** (least significant bit) and **vec** (rest of the vector), and the recognizer **bitvp**.

III. MODELLING AND VERIFYING REPLICATED DEVICES

In this paragraph, we give recursive models for regular architectures. Recursion is well-suited for describing the duplication of a same hardware motif. Throughout this section, recursive functional forms are associated with the outputs of replicated structures :

- the parameters are the inputs of the system, and carry propagation is translated by the updating of the corresponding parameter(s) with the value(s) computed by the function(s) that express this propagation,

- if the output is vectorial, the function builds this vector by means of `bitv`, which appends a bit to a bit-vector.

III.1 A recursive model for one-dimensional replicated circuits

Here, we give recursive functional models for representing the output(s) of one-dimensional regular architectures. These devices usually take as inputs several bit-vectors of identical length N, which is given as a `generic` parameter of the description. Since we deal with bit-vectors of the same length, only one of them is used as recursion parameter in our definitions. In the following, we use the term "carry" for describing any boolean result computed in the ith module and passed to the (i+1)th one.

Notation : the vectorial inputs are denoted $VI_1, ..., VI_j$, and the scalar inputs (which include the propagated carries) are denoted $SI_1, ..., SI_k$.

First, we give the function which corresponds to a vectorial output VO_i, $1 \leq i \leq i_{VO}$, where i_{VO} is usually $\leq j$. The function $REC\text{-}F_{VOi}$ builds the bit-vector formed by the result of F_{VOi} (the function associated with the corresponding boolean output of the basic cell), catenated to the result of the recursive call to $REC\text{-}F_{VOi}$ with the vectorial inputs of one element less and the scalar inputs updated by the functions $F_{SI1}, \dots$, F_{SIk} (which express the carry propagation) :

```
REC-F_VOi(VI_1, ..., VI_j, SI_1, ..., SI_k) =def
  if bitvp(VI_1)          --- if VI_1 is a bit-vector
  then if VI_1 = (btm)             --- if VI_1 is empty, the result
       then F_SI(SI_1, ..., SI_k)  --- is a function of SI_1, ..., SI_k.
       --- if VI_1 is not empty, we recursively call REC-F_VOi :
       else bitv(F_VOi(bit(VI_1), ..., bit(VI_j), SI_1, ..., SI_k),
                 REC-F_VOi(vec(VI_1), ..., vec(VI_j),
                           F_SI1(bit(VI_1), ..., bit(VI_j), SI_1,..., SI_k),
                           ...
                           F_SIk(bit(VI_1),..., bit(VI_j), SI_1,..., SI_k)))
  else F_SI(SI_1, ..., SI_k)
```

The model which is associated with **a scalar output** SO_i, $1 \leq i \leq i_{SO}$, is similar, except that it does not build a vectorial result :

```
REC-F_SOi(VI_1, ..., VI_j, SI_1, ..., SI_k) =def
  if bitvp(VI_1)          --- if VI_1 is a bit-vector
  then if VI_1 = (btm)             --- if VI_1 is empty, the result
       then F_SI(SI_i)             --- is a function of SI_i
       --- if VI_1 is not empty, we recursively call REC-F_SOi :
       else REC-F_SOi(vec(VI_1), ..., vec(VI_j),
                      F_SI1(bit(VI_1), ..., bit(VI_j), SI_1, ..., SI_k),
                      ...
                      F_SIk(bit(VI_1), ..., bit(VI_j), SI_1, ..., SI_k))
  else F_SI(SI_i)
```

Example : The description of the ripple-carry adder (Figure 1) has one vectorial output `z`. It is expressed by the following recursive function `RCA-z` (given in the Boyer-Moore syntax), which is of the form of $\text{REC-F}_{\text{VOi}}$ where `A` and `B` are the vectorial inputs and `c` is a scalar input (the carry to be propagated) :

```
(defn RCA-z (A B c)
  (if (bitvp A)
      (if (equal A (btm))
          (bitv c (btm))
          (bitv (fulladd-fout (bit A) (bit B) c)
                (RCA-z (vec A)
                       (vec B)
                       (fulladd-fcarry (bit A) (bit B) c))))
      (bitv c (btm))))
```

where `fulladd-fout` and `fulladd-fcarry` are the non-recursive functions associated with the outputs `fout` and `fcarry` of the component `fulladd`.

III.2 A recursive model for two-dimensional replicated circuits

Now, let us focus on two-dimensional regular structures. More precisely, we are interested in parallel systems which are built from N instances of the same row, where each row is also a replicated device. Thus, the outputs of the basic row are represented by functional forms which correspond to the templates of § III.1.

Here, we give a recursive model for representing the output of the whole parallel architecture; this function will be defined in terms of the $\text{REC-F}_{\text{VOi}}$, $1 \le i \le i_{VO}$.

Notation : the vectorial inputs are denoted $\text{VI}_1, \ldots, \text{VI}_p$, and the scalar inputs are denoted $\text{SI}_1, \ldots, \text{SI}_q$. Typically, $p > j$ where j is the index used in § III.1. A necessary condition is $i_{VO} \le j$. Usually, the architecture is such that $i_{VO} = j-1$, $q = k-1$, and $p = j+1$ (i.e. VI_p expresses the second dimension). In order to simplify the model given below, we will assume these equalities (but it can easily be transformed into a more general one).

The following function F_{par} models the output of such a parallel circuit :

```
Fpar(VI1, ..., VIp, SI1, ..., SIq) =def
   if bitvp(VIp)        --- if VIp is a bit-vector
   then if VIp = (btm)          --- if VIp is empty, the result is
                                --- output by the last component
        then Last-comp(VI1, ..., VIp-1, SI1, ..., SIq)
        --- if VIp is not empty, we recursively call Fpar :
        else bitv(Fb(bit(VI2), ..., bit(VIp-1)),
                  Fpar(VI1,
                       REC-FVO1(VI1, ..., bit(VIp), SI1, ..., SIq),
                       ...,
                       REC-FVOivo(VI1, ..., bit(VIp), SI1, ..., SIq),
                       vec(VIp), SI1, ..., SIq))
   else Last-comp(VI1, ..., VIp-1, SI1, ..., SIq)
```

This function builds the bit-vector formed by the result of F_b(bit(VI_2), ..., bit(VI_{p-1})) where F_b is a boolean function (the function associated with the corresponding output of the basic row), catenated to the result of the recursive call to F_{par} with the vectorial input VI_p of one element less and the vectorial inputs updated by the functions REC-F_{VO1}, ..., REC-$F_{VOi_{vo}}$ (which express, if needed, the carry propagation through the rows).

Remark : Last-comp is a vectorial function which represents the last row, in particular it can be different from the other ones.

An illustrative example of this modelling method can be found in § V.2.

III.3 The induction-based proof method

1. As far as **one-dimensional circuits** are concerned, the proof process consists in verifying the equivalence between each function REC-F_{VOi} or REC-F_{SOi} (or a combination of these functions) and a function which expresses the arithmetic specification. In order to (formally) compare the results computed by these functions, we have to abstract the first one to the arithmetic level, using the function bv-to-nat which converts bit-vectors into natural numbers, starting from the LSB.

The theorem to be proved generally states the equivalence between the result of the function REC-F_{VOi} converted into a natural number and a specification function, denoted below $SPEC_{VOi}$. This property is generally given in terms of particular initial values for the scalar inputs, referred to as IV-SI_1, ..., IV-SI_k :

```
bitvp(VI1) and ... and bitvp(VIj)      --- the VIi are bit-vectors
and size(VI1) = size(VI2)              --- the vectors have the same length
... and size(VI1) = size(VIj)
                          ⇓                                                  Th1
bv-to-nat(REC-FVOi(VI1, ..., VIj, IV-SI1, ..., IV-SIk))
= SPECVOi(bv-to-nat(VI1), ..., bv-to-nat(VIj))
```

Remarks : • In the case where we have to reason on REC-F_{SOi}, we use the same statement, except that we do not have to call on the function bv-to-nat since REC-F_{SOi} gives a boolean result.

• If the property to be proved is an equivalence between a combination of the results of the functions REC-F_{VO1}, ..., REC-F_{VOj} and a specification expression, we use the same kind of theorem except that the term bv-to-nat(REC-F_{VOi}(VI_1, ..., VI_j, IV-SI_1,..., IV-SI_k)) is replaced by a term of the form F(bv-to-nat(REC-F_{VO1}(VI_1, ..., IV-SI_k)), ..., bv-to-nat(REC-F_{VOj}(VI_1, ..., IV-SI_k))).

Example : The correctness theorem for the circuit of Figure 1 is the theorem proof-of-RCA-z which verifies that this device outputs the sum of the two bit-vectors A and B, provided that the carry-in is set to false :

```
(prove-lemma proof-of-RCA-z (rewrite)
   (implies (and (bitvp A) (bitvp B) (equal (size A) (size B)))
            (equal (bv-to-nat (RCA-z A B f))
                   (plus (bv-to-nat A) (bv-to-nat B)))))
```

We will see in section V that other significant examples can be verified using the same modelling and proof methodology.

2. With respect to **two-dimensional architectures**, the verification process is hierarchically decomposed into sub-proofs : first we verify the correctness of the basic row, and then it is possible to validate the whole device.

The first necessary step consists in giving pieces of information about the size(s) of the bit-vector(s) output by the one-dimensional row. One theorem per row output indicates the size of this vectorial output in terms of the input size(s). Then, the correctness of this row is verified using one or several theorems of the form of Th_1, provided that they have previously been generalized. Finally, a theorem of the form of Th_2 below allows to validate the two-dimensional structure :

```
bitvp(VI_1) and bitvp(VI_p)      --- VI_1 and VI_p are bit-vectors
and size(VI_1) = size(VI_p)      --- They have the same length
                         ⇓
bv-to-nat(F_par(VI_1, F_init1(VI_1, VI_p), ...,
              F_initp-2(VI_1, VI_p), VI_p, IV-SI_1, ..., IV-SI_q))
= SPEC(bv-to-nat(VI_1), ..., bv-to-nat(VI_p))
```
Th_2

where - SPEC corresponds to the specification,
- F_{init1} is the function which computes the initial value of VI_2, ..., and $F_{initp-2}$ is the function which computes the initial value of VI_{p-1},
- and $IV\text{-}SI_i$, $1 \leq i \leq q$, is the initial value of SI_i.

Here, as well as in the case of one-dimensional devices, the presence of particular values for the incoming carries implies that an inductive proof is not feasible. Such a theorem must be generalized, this is the purpose of section IV.

IV. THE GENERALIZATION METHOD

IV.1 Motivation

In the fields of software as well as hardware verification, generalization is often necessary as soon as induction-based techniques are applied. For instance, Manna and Waldinger describe the problem of "generalization of specifications" in the framework of program synthesis [16]. Finding general-purpose generalization methods that can be fully mechanized is almost impossible. Some interesting heuristics, that require user-interaction, have been proposed. Among them, let us recall the following approaches :

- J Moore, one of the authors of the Boyer-Moore prover, proposed an interesting generalization heuristics for recursive functions with accumulating parameters [15]. However, there is a step where the user has to "guess" the term by which a certain variable must be replaced. In fact, this heuristics has not been implemented in the Boyer-Moore system.
- R.Aubin also worked on such generalization heuristics. With respect to the problem of verifying the equivalence between recursive functions and corresponding iterative ones, he gave a method called "indirect generalization" [2]. This method introduces, in the expression to be proved, a new function call

associated with its neutral element, and then generalizes this constant. Here also, the appropriate function has to be intuitively determined by the user.

Our goal is not to propose a general-purpose technique. We give a specialized method for our problem, and this special-purpose algorithm can be mechanized. A prototype implementation has been included in PREVAIL™.

IV.2 Generalization algorithm

The algorithm below is devoted to regular replicated architectures. Starting from a theorem of the form of Th_1 or Th_2, this method yields a more general lemma which is provable by induction. The generalization algorithm is :

```
Unfold once the definition associated with the implementation (i.e. REC-F_VOi or REC-F_SOi or F_par) in the left-hand side of the equality ;

While the new expression includes terms of the form bit(VI_n) (or bit(GVI_n)) do
    Perform a case analysis on the value of bit(VI_n) :   bit(VI_n) = f
                                                          bit(VI_n) = t
    For each case do
        Simplify the resulting equality ;
        Unfold once the definition of "bv-to-nat" ;
        Apply simplification rules ;
        Generalize the term vec(VI_n) into GVI_n ;
    EndFor ;
EndWhile ;

From the set of expressions obtained from the previous case analysis, deduce a single (possibly conditional) expression E ;

If the implementation function corresponds to a two-dimensional architecture
Then  use theorems Th_1 to transform the left-hand side of this equality (and the
      right-hand side accordingly) :

E is of the form :
  bv-to-nat(F_par(VI_1, REC-F_VO1(VI_1, VI_2', ..., vi_p', SI_1, ..., SI_q),
                  ..., REC-F_VOivo(VI_1, VI_2', ..., vi_p', SI_1, ..., SI_q),
                  VI_p, SI_1, ..., SI_q))
  = G(bv-to-nat(VI_1),
      F_1(bv-to-nat(VI_1), ..., bv-to-nat(VI_p-1'), vi_p', SI_1, ..., SI_q),
      ...
      F_ivo(bv-to-nat(VI_1), ..., bv-to-nat(VI_p-1'), vi_p', SI_1,..., SI_q),
      bv-to-nat(VI_p))

and theorems Th_1 give equalities of the form :
  bv-to-nat(REC-F_VOi(VI_1, ..., VI_p-1, vi_p, SI_1, ..., SI_q))
  = SPEC_VOi(bv-to-nat(VI_1), ..., bv-to-nat(VI_p-1), vi_p, SI_1, ..., SI_q)
where vi_p plays the role of bit(VI_p).
```

By replacing $\mathtt{VI}_2$ by $\mathtt{VI}_2$', ..., $\mathtt{VI}_{p-1}$ by $\mathtt{VI}_{p-1}$', $\mathtt{vi}_p$ by $\mathtt{vi}_p$', in these equalities, we get equalities such as :

```
bv-to-nat(REC-F_voi(VI_1, ..., VI_p-1', vi_p', SI_1, ..., SI_q))
= SPEC_voi(bv-to-nat(VI_1), ...,bv-to-nat(VI_p-1'),vi_p',SI_1, ...,SI_q)
```

which is precisely of the form of

```
    F_i(bv-to-nat(VI_1), ...,bv-to-nat(VI_p-1'),vi_p', SI_1, ..., SI_q)
```

It allows to rewrite E into the following expression :

```
bv-to-nat(F_par(VI_1, REC-F_vo1(VI_1, VI_2', ..., vi_p', SI_1, ..., SI_q),
                ..., REC-F_voivo(VI_1, VI_2', ..., vi_p', SI_1, ..., SI_q), VI_p,
                SI_1, ..., SI_q))
= G(bv-to-nat(VI_1),
    bv-to-nat(REC-F_vo1(VI_1, ..., VI_p-1', vi_p', SI_1, ..., SI_q))),
    ...,
    bv-to-nat(REC-F_voivo(VI_1, ..., VI_p-1', vi_p', SI_1, ..., SI_q))),
    bv-to-nat(VI_p))
```

which is finally generalized into :

```
bv-to-nat(F_par(VI_1, X_1, ..., X_ivo, VI_p, SI_1, ..., SI_q))
= G (bv-to-nat(VI_1), bv-to-nat(X_1), ..., bv-to-nat(X_ivo),
     bv-to-nat(VI_p))
```

Result := this expression ;

Else Result := E;

EndIf.

Example : As far as the ripple-carry adder is concerned, the algorithm stops at the end of the *While* loop. Starting from :

```
(prove-lemma proof-of-RCA-z (rewrite)
   (implies (and (bitvp A) (bitvp B) (equal (size A) (size B)))
            (equal (bv-to-nat (RCA-z A B f))
                   (plus (bv-to-nat A) (bv-to-nat B)))))
```

with

```
(defn bv-to-nat (x)
   (if (bitvp x)
       (if (equal x (btm))
           0
           (plus (if (bit x) 1 0) (times 2 (bv-to-nat (vec x)))))
   0))
```

the generalization process is :

Assuming that we are not in the basis case, and unfolding the definition of RCA-z *gives :*

```
(equal (bv-to-nat
          (bitv (xor f (xor (bit A) (bit B)))
                (RCA-z (vec A) (vec B)
                       (or (and (bit A) (bit B))
                           (or (and (bit A) f) (and (bit B) f))))))
       (plus (bv-to-nat A) (bv-to-nat B)))
```

which simplifies into :

```
(equal (bv-to-nat
              (bitv (xor (bit A) (bit B))
                     (RCA-z (vec A) (vec B) (and (bit A) (bit B)))))
       (plus (bv-to-nat A) (bv-to-nat B)))
```

and the case analysis is :

1. If (bit a) *= true*

```
(equal (bv-to-nat
             (bitv (not (bit b)) (RCA-z (vec A) (vec B) (bit B))))
       (plus (add1 (times 2 (bv-to-nat (vec A)))) (bv-to-nat B)))
```

i.e.

```
(equal (bv-to-nat (bitv (not (bit b)) (RCA-z GA (vec B) (bit B))))
       (plus (add1 (times 2 (bv-to-nat GA))) (bv-to-nat B)))
```

1.1. If (bit b) *= true*

```
    (equal (bv-to-nat (bitv f (RCA-z GA (vec B) t)))
           (plus (add1 (times 2 (bv-to-nat GA)))
                 (add1 (times 2 (bv-to-nat (vec B))))))
```

i.e.

```
    (equal (times 2 (bv-to-nat (RCA-z GA (vec B) t)))
           (times 2 (add1 (plus (bv-to-nat GA))
                                (bv-to-nat (vec B))))))
```

which is simplified and generalized into

```
    (equal (bv-to-nat (RCA-z GA GB t))
           (add1 (plus (bv-to-nat GA)) (bv-to-nat GB))))
```

1.2. If (bit b) *= false*

```
    (equal (bv-to-nat (bitv t (RCA-z GA (vec B) f))))
           (plus (add1 (times 2 (bv-to-nat GA)))
                 (times 2 (bv-to-nat (vec B)))))
```

i.e.

```
    (equal (add1 (times 2 (bv-to-nat (RCA-z GA (vec B) f))))
           (add1 (plus (times 2 (bv-to-nat GA))
                       (times 2 (bv-to-nat (vec B))))))
```

which is simplified and generalized into

```
    (equal (bv-to-nat (RCA-z GA GB f))
           (plus (bv-to-nat GA) (bv-to-nat GB))
```

2. If (bit a) *= false*

```
(equal (bv-to-nat (bitv (bit b) (RCA-z (vec A) (vec B) f)))
       (plus (times 2 (bv-to-nat (vec A))) (bv-to-nat B)))
```

i.e.

```
(equal (bv-to-nat (bitv (bit b) (RCA-z GA (vec B) f)))
       (plus (times 2 (bv-to-nat GA)) (bv-to-nat B)))
```

Similarly, we get in that case :

2.1. If (bit b) *= true*

```
    (equal (bv-to-nat (RCA-z GA GB f))
           (plus (bv-to-nat GA) (bv-to-nat GB))
```

2.2. If (bit b) *= false*

```
    (equal (bv-to-nat (RCA-z GA GB f))
           (plus (bv-to-nat GA) (bv-to-nat GB)))
```

Then, from sub-cases 1.2, 2.1 and 2.2, we have :

```
(equal (bv-to-nat (RCA-z GA GB f))
       (plus (bv-to-nat GA) (bv-to-nat GB)))
```

and from sub-case 1.1, we have :

```
(equal (bv-to-nat (RCA-z GA GB t))
       (add1 (plus (bv-to-nat GA) (bv-to-nat GB))))
```

Thus, we deduce the complete generalized theorem :

```
(prove-lemma proof-of-RCA-z-gen (rewrite)
   (implies (and (bitvp GA) (bitvp GB) (boolp c)
                 (equal (size GA) (size GB)))
            (equal (bv-to-nat (RCA-z GA GB c))
                   (if c (add1 (plus (bv-to-nat GA) (bv-to-nat GB)))
                         (plus (bv-to-nat GA) (bv-to-nat GB))))))
```

V. APPLICATION TO THE TPCD BENCHMARKS

Among the devices to which this technique applies, some of them have been proposed as *TPCD benchmarks*, or parts of these benchmarks [14]. Apart from the N-bit ripple-carry adder, some sub-modules of the Min-Max circuit, as well as the parallel multiplier can be given as illustrative examples.

V.1 Min-Max

A specification of this benchmark has been proposed for the IFIP WG 10.2 International Workshop on "Applied Formal Methods for Correct VLSI Design" [22]. We have designed an implementation (with arithmetic modules for unsigned bit-vectors only) and we have realized a hierarchical proof of this circuit [18].

The specification is that the Min-Max unit has 3 boolean control signals CLEAR, RESET and ENABLE. The unit produces an output sequence OUT at the same rate as IN :

- if CLEAR is *true*, then OUT equals 0, independent of the other control signals,
- if CLEAR is *false* and ENABLE is *false*, then OUT equals the last value of IN before ENABLE became *false*,
- if CLEAR is *false*, ENABLE is *true*, and RESET is *true*, then OUT follows IN,
- if RESET becomes *false*, then OUT holds, on each time point t, the mean value of the maximum and minimum value of IN until that time point, since RESET became *false*.

The most important component of our implementation is a sub-module "MeanValue" which is supposed to correspond to the last part of the specification : OUT *holds, on each time point t, the mean value of the maximum and minimum value of* IN *until that time point, since* **RESET** *became false*. This device is depicted on Figure 2. PASTMAX and PASTMIN are two registers, and the other components are N-bit arithmetic modules :

- GREAT_N and LESS_N implement the ">" and "<" comparisons on N-bit vectors,
- MUX_N multiplexes two N-bit vectors according to a control bit,
- RCA is the ripple-carry adder ,
- and RIGHTSHIFT shifts a N-bit vector to the right, i.e. divides the corresponding natural number by 2.

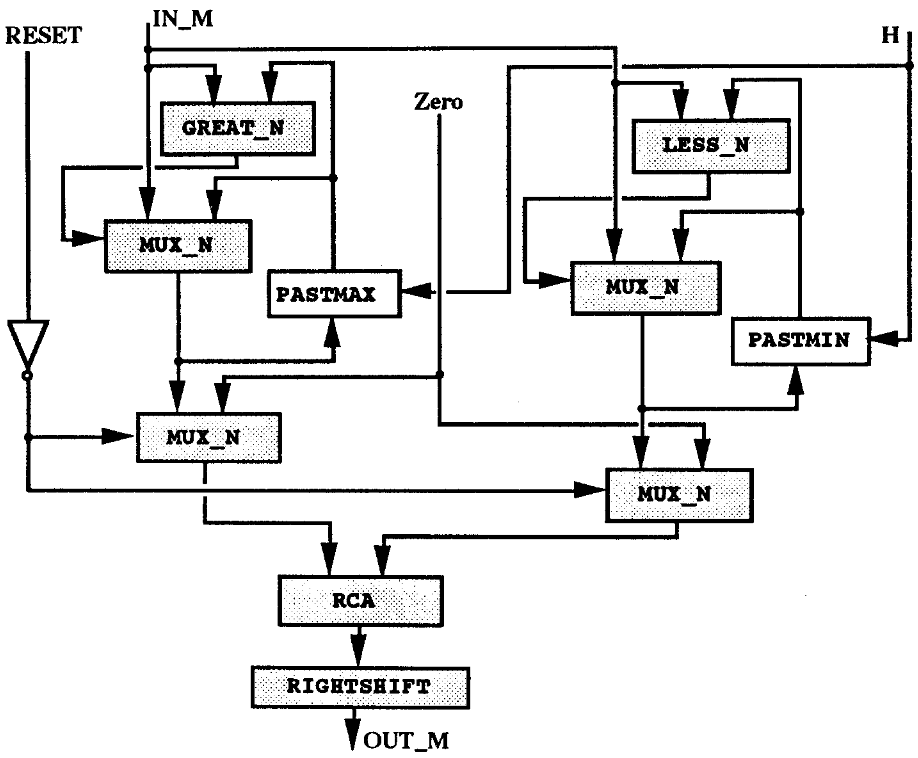

FIGURE 2 : MEANVALUE MODULE

Because of the lack of place, we give neither all the pictures of the sub-modules nor the VHDL descriptions. We will only describe the Boyer-Moore verification of the most significant components. The Boyer-Moore code is automatically generated from the VHDL descriptions, with a limited user interaction (through a menu-directed interface).

1. The rightshift component.

Here, we give a more elaborate version of this device than in [18]. This circuit inputs a N-bit vector A and a boolean signal c, and outputs a N-bit vector Z which corresponds to A shifted to the right, and the most significant bit of which takes the value of c. Thus, the Boyer-Moore function associated with Z is :

```
(defn rightshift-Z (A c)
  (if (bitvp A)
      (if (equal A (btm))
          (btm)
          (if (equal (vec A) (btm))
              (bitv c (btm))
              (bitv (bit (vec A)) (rightshift-Z (vec A) c))))
      (btm)))
```

We have to verify that, when c equals `false`, the result corresponds to `(bv-to-nat A)` divided by 2, i.e. :

```
(prove-lemma proof-of-rightshift-Z (rewrite)
  (implies (bitvp A)
           (equal (bv-to-nat (rightshift-Z A f))
                  (quotient (bv-to-nat A) 2))))
```

Since c is not propagated through the shifter cells, generalization is not mandatory. However, this lemma can be generalized into the following one :

```
(prove-lemma proof-of-rightshift-Z-gen (rewrite)
  (implies (and (bitvp A) (not (equal A (btm))) (boolp c))
           (equal (bv-to-nat (rightshift-Z A c))
                  (if c
                      (plus (quotient (bv-to-nat A) 2)
                            (exp 2 (sub1 (size A))))
                      (quotient (bv-to-nat A) 2)))))
```

2. The modules GREAT_N and LESS_N.

Each of these devices inputs two N-bit vectors A and B, and a carry-in X_IN, and compares the bit-vectors, starting from the least significant bit. The carry is propagated up to the last cell, where it finally represents the result of the comparison. Figure 3 depicts the component GREAT_N, the other one is rather similar. We give the proof of this module GREAT_N, the verification of LESS_N is processed similarly.

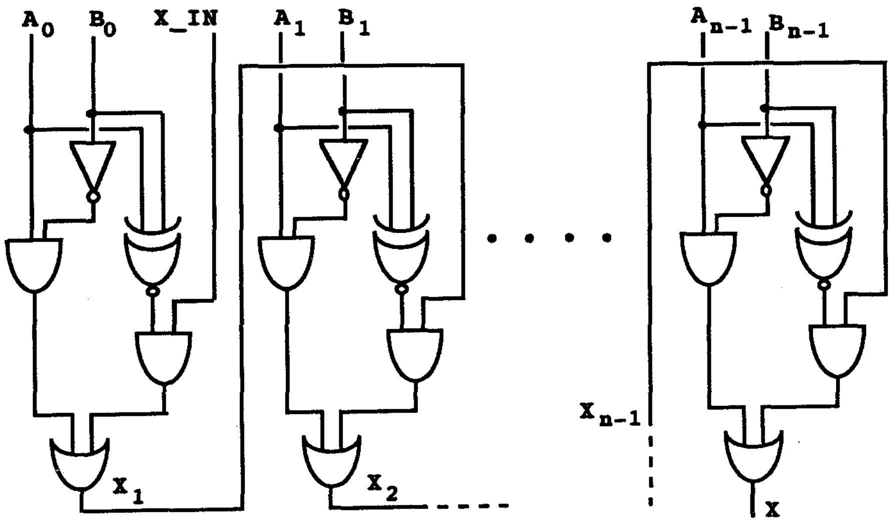

FIGURE 3 : GREAT_N MODULE

This circuit has a boolean output X, and the corresponding Boyer-Moore function is :

```
(defn great_n-X (A B X_IN)
  (if (bitvp A)
      (if (equal A (btm))
          X_IN
          (great_n-X (vec A) (vec B)
                     (or (and (bit A) (not (bit B)))
                         (and (eqv (bit A) (bit B)) X_IN))))
      X_IN))
```

Then, we have to verify that this device implements correctly the associated comparison, provided that the input carry equals `false`. Our system generates the following theorem :

```
(prove-lemma proof-of-great_n-X (rewrite)
   (implies (and (bitvp A) (bitvp B) (equal (size A) (size B)))
           (equal (great_n-X A B f)
                 (greaterp (bv-to-nat A) (bv-to-nat B)))))
```

Here, generalization is mandatory. The generalization algorithm generates 4 sub-cases and finally produces the following generalized lemma :

```
(prove-lemma proof-of-great_n-X-gen (rewrite)
   (implies (and (bitvp A) (bitvp B)
                (equal (size A) (size B)) (boolp x))
           (equal (great_n-X A B x)
                 (if x
                     (or (greaterp (bv-to-nat A) (bv-to-nat B))
                         (equal (bv-to-nat A) (bv-to-nat B)))
                     (greaterp (bv-to-nat A) (bv-to-nat B))))))
```

V.2 Parallel multiplier

This second benchmark will illustrate our methodology for two-dimensional architectures. This device is a combinational parallel multiplier. Many significant methodologies have been developed for the verification of parallel array multipliers. The most efficient ones are based on constraint logic programming [21], or have been inspired from the development of tautology-checking techniques based on BDD-like representations [8], [1]. However, R.Bryant has shown that the size of the BDD representing multiplication grows exponentially in the number of input bits [7]. Even though [8] and [1] try to overcome this problem, they do not take advantage of the fact that most of these circuits are completely (or at least partially) regular.

The design that is proposed in [14] exactly corresponds to the circuit that is referred to as the "Braun's array multiplier" in [12]. The correspondence between the basic cell used in the TPCD design and the fulladder-based module used in [12] is given by Figure 4 below.

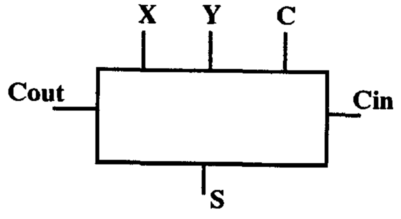

Figure 4a. TPCD module

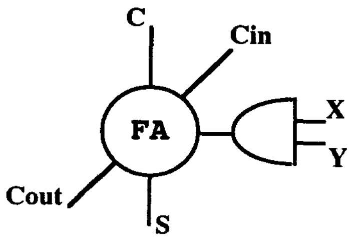

Figure 4b. FullAdder-based module

FIGURE 4 : CORRESPONDENCE BETWEEN BASIC MODULES

A 4-bit version of the Braun multiplier is depicted by Figure 5. A generic N-bit VHDL description and a detailed Nqthm proof can be found in [20]. Here we propose an outline of this verification. This device is made of a succession of similar rows, and the last row is a ripple-carry adder. Its hierarchical proof consists in validating the basic row, the ripple-carry adder (proof already presented), and then the whole multiplier.

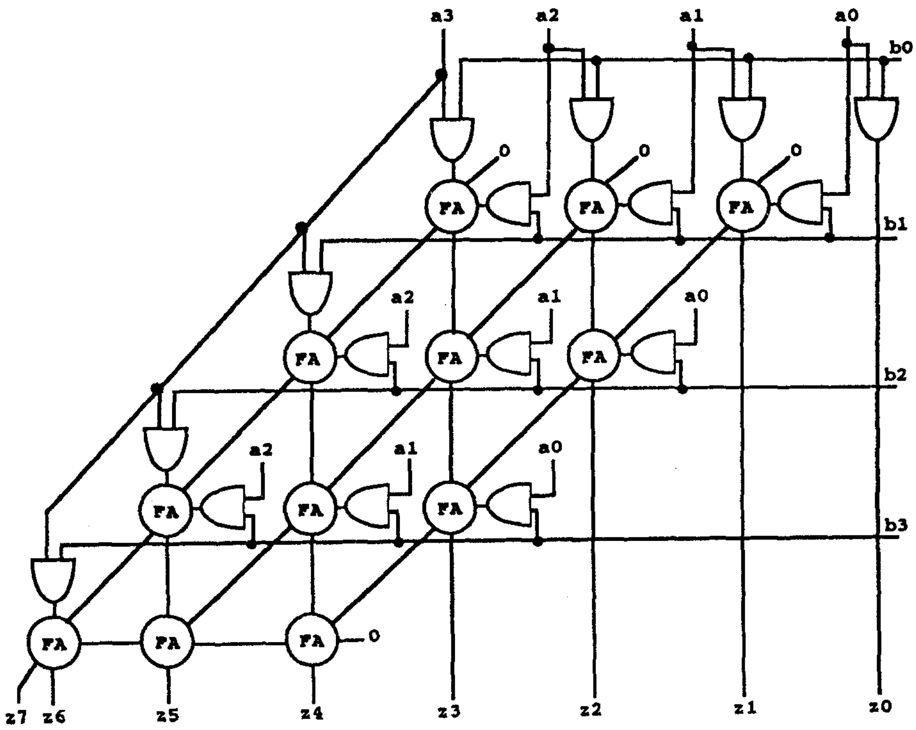

FIGURE 5 : 4-BIT BRAUN MULTIPLIER

Two functions are associated with the two vectorial outputs of the basic row. The case where `(vec A) = (btm)` must be processed separately (irregularity on the last cell) :

```
(defn Row-x_out (A X Y b)
  (if (bitvp A)
      (if (equal A (btm))
          (btm)
          (if (equal (vec A) (btm))
              (bitv (and (bit A) b) (btm))
              (bitv (fulladd-fout (bit X) (bit Y) (and (bit A) b))
                     (Row-x_out (vec A) (vec X) (vec Y) b))))
      (btm)))
(defn Row-y_out (A X Y b)
  (if (bitvp A)
      (if (equal A (btm))
          (btm)
          (if (equal (vec A) (btm))
              (btm)
              (bitv (fulladd-fcarry (bit X) (bit Y) (and (bit A) b))
                     (Row-y_out (vec A) (vec X) (vec Y) b))))
      (btm)))
```

The following function `Mult-Z` corresponds to the vectorial output of the multiplier, where `left-shift` is the function which specifies a left-shifter, i.e. a multiplication by two (in our system, its definition and associated properties are included in a library of pre-proven and re-usable basic components) :

```
(defn Mult-Z (A X Y B)
  (if (bitvp B)
      (if (equal B (btm))
          (RCA-z X (left-shift Y) f)
          (bitv (bit X)
                (Mult-Z A
                        (Row-x_out A (vec X) Y (bit B))
                        (Row-y_out A (vec X) Y (bit B))
                        (vec B))))
      X))
```

The first verification step corresponds to the following theorems `size-of-Row-x_out` and `size-of-Row-y_out` :

```
(prove-lemma size-of-Row-x_out (rewrite)
  (equal (size (Row-x_out A X Y b)) (size A)))))

(prove-lemma size-of-Row-y_out (rewrite)
  (equal (size (Row-y_out A X Y b)) (sub1 (size A))))
```

Then, we verify the functional **specification of the basic row** which depends on the initial value of `b`. It is given by the table below, where X' and Y' represent the bit-vectors output by the row :

value of b	**bv-to-nat (X') + 2 bv-to-nat (Y')**
f	bv-to-nat (x) + bv-to-nat (y)
t	bv-to-nat (x) + bv-to-nat (y) + bv-to-nat (a)

The Boyer-Moore theorem which translates this specification is of the form of *Th1* where we consider the appropriate combination of the row outputs :

```
(prove-lemma proof-of-Row-x_out-y_out (rewrite)
  (implies (and (bitvp X) (bitvp Y) (bitvp A) (boolp b)
                (equal (size Y) (size X))
                (equal (size X) (sub1 (size A))))
           (equal (plus (bv-to-nat (Row-x_out A X Y b))
                        (times 2 (bv-to-nat (Row-y_out A X Y b))))
                  (plus (plus (bv-to-nat X) (bv-to-nat Y))
                        (if b (bv-to-nat A) 0)))))
```

After this proof, we should try to verify the **correctness of the multiplier**, i.e. the lemma `proof-of-Mult-Z` below :

```
(prove-lemma proof-of-Mult-Z (rewrite)
  (implies (and (bitvp A) (bitvp B)
                (equal (size B) (size A)))
           (equal (bv-to-nat (Mult-Z A
                                     (init_and A (bit B))
                                     (all-zeros (sub1 (size A)))
                                     (vec B)))
                  (times (bv-to-nat A) (bv-to-nat B)))))
```

where the functions `init`$_{and}$ and `all-zeros` are used to initialize the vectorial inputs `X` and `Y` : the initial value of `X` is the result of "and-ing" each bit of `A` with the first bit of `B`, and the initial value of `Y` is a vector each bit of which is equal to `false`.

The lemma `proof-of-Mult-Z` is generalized into the theorem `proof-of-Mult-Z-gen`:

```
(prove-lemma proof-of-Mult-Z-gen (rewrite)
  (implies (and (bitvp A) (bitvp B) (bitvp X) (bitvp Y)
                (equal (size Y) (sub1 (size A)))
                (equal (size X) (size A)) (lessp (size B) (size A)))
           (equal (bv-to-nat (Mult-Z A X Y B))
                  (plus (times 2 (times (bv-to-nat A) (bv-to-nat B)))
                        (plus (times 2 (bv-to-nat Y))
                              (bv-to-nat X))))))
```

The first phase of the generalization algorithm generates 8 sub-cases, from which the following equalities are deduced :

```
(equal (bv-to-nat (Mult-Z A
                          (Row-x_out A (all-zeros (sub1 (size A)))
                                       (all-zeros (sub1 (size A))) c)
                          (Row-y_out A (all-zeros (sub1 (size A)))
                                       (all-zeros (sub1 (size A))) c)
                          B))
       (plus (times 2 (times (bv-to-nat A) (bv-to-nat B)))
             (if c (bv-to-nat A) 0)))
```

and

```
(equal (bv-to-nat (Mult-Z A
                          (Row-x_out A (vec A)
                                       (all-zeros (sub1 (size A))) c)
                          (Row-y_out A (vec A)
                                       (all-zeros (sub1 (size A))) c)
                          B))
       (plus (plus (times 2 (times (bv-to-nat A) (bv-to-nat B)))
                   (bv-to-nat (vec A)))
             (if c (bv-to-nat A) 0)))
```

With regard to the first one, an ad hoc instantiation of theorem `proof-of-Row-x_out-y_out` gives :

```
(equal (plus (bv-to-nat (Row-x_out A (all-zeros (sub1 (size A)))
                                     (all-zeros (sub1 (size A))) c))
             (times 2 (bv-to-nat
                        (Row-y_out A (all-zeros (sub1 (size A)))
                                     (all-zeros (sub1 (size A)))
                                     c))))
       (plus (plus (bv-to-nat (all-zeros (sub1 (size A))))
                   (bv-to-nat (all-zeros (sub1 (size A)))))
             (if c (bv-to-nat A) 0)))
```

i.e.

```
(equal (plus (bv-to-nat (Row-x_out A (all-zeros (sub1 (size A)))
                                     (all-zeros (sub1 (size A))) c))
             (times 2 (bv-to-nat
                        (Row-y_out A (all-zeros (sub1 (size A)))
                                   (all-zeros (sub1 (size A)))
                                   c))))
       (if c (bv-to-nat A) 0))
```

Thus, replacing `(if c (bv-to-nat A) 0)` by the equivalent expression in the first equality above produces :

```
(equal (bv-to-nat (Mult-Z A
                            (Row-x_out A (all-zeros (sub1 (size A)))
                                         (all-zeros (sub1 (size A))) c)
                            (Row-y_out A (all-zeros (sub1 (size A)))
                                         (all-zeros (sub1 (size A))) c)
                            B))
       (plus (times 2 (times (bv-to-nat A) (bv-to-nat B)))
             (plus (bv-to-nat (Row-x_out A (all-zeros (sub1 (size A)))
                                           (all-zeros (sub1 (size A))) c))
                   (times 2
                          (bv-to-nat (Row-y_out A
                                                (all-zeros (sub1 (size A)))
                                                (all-zeros (sub1 (size A)))
                                                c))))
```

and generalizing the terms `(Row-x_out ...)` and `(Row-y_out ...)` finally gives :

```
(equal (bv-to-nat (Mult-Z A X Y B))
       (plus (times 2 (times (bv-to-nat A) (bv-to-nat B)))
             (plus (bv-to-nat X) (times 2 (bv-to-nat Y)))))
```

Similarly, for the second one, an ad hoc instantiation of theorem `proof-of-Row-x_out-y_out` gives :

```
(equal (plus (bv-to-nat (Row-x_out A (vec A)
                                     (all-zeros (sub1 (size A))) c))
             (times 2 (bv-to-nat (Row-y_out A (vec A)
                                             (all-zeros (sub1 (size A)))
                                             c))))
       (plus (plus (bv-to-nat (vec A))
                   (bv-to-nat (all-zeros (sub1 (size A)))))
             (if c (bv-to-nat A) 0)))
```

which finally produces, after generalization :

```
(equal (bv-to-nat (Mult-Z A X Y B))
       (plus (times 2 (times (bv-to-nat A) (bv-to-nat B)))
             (plus (bv-to-nat X) (times 2 (bv-to-nat Y)))))
```

Both equalities produce the same expression which becomes, with the appropriate hypotheses, the lemma `proof-of-Mult-Z-gen.`

Now, let us compare the efficiency of our approach with results given in significant articles referenced at the beginning of section V.2. The following table gives the number of theorems to be proved and the total CPU times (on a SUN Sparc2, with 32 Mb of memory) for the complete Boyer-Moore proof of this Braun multiplier, as well as for the verification of a simpler multiplier architecture given as example in [8].

	Simple N-bit multiplier	N-bit BRAUN multiplier
Number of theorems	3	5
Total CPU time	14.6 seconds	99.2 seconds

In the table below, we recall some of the experimental results presented in the referenced papers :

Paper	Multiplier size	Proof time
[21]	44-bit	**~100s** on a Sun 3/260
[1]	32-bit	output #31 in **533mn**, and output #15 in **25h**, on a IBM 6000/320 (with 256 Mb of memory)
[8]	16-bit (C6288, ISCAS'85 bench.)	**~40mn** on a Sun 3/60 (with 12 Mb of memory)

The comparison of these tables demonstrates that our technique allows the verification of N-bit architectures within satisfying CPU times.

VI. CONCLUSION

We have proposed an inductive proof methodology devoted to the formal verification of one-dimensional, as well as two-dimensional, replicated structures. This method has been mechanized in the Boyer-Moore logic. We have also described an associated generalization technique for transforming the properties to be verified, before the proof process. The automatic translator between VHDL and Boyer-Moore and the generalization algorithm are being implemented. The examples that illustrate this paper demonstrate the feasibility and the efficiency of the method for validating generic N-bit devices.

Our proof technique requires a particular design methodology; it is not applicable starting from a simple netlist of such a parallel circuit. The translation task between VHDL and Boyer-Moore is feasible if precise description rules have been respected. The illustrative examples show that the circuit descriptions must be hierarchically organized, and the regularity must be expressed by `for ... generate` and `for ... loop` statements. Future work will aim at defining a "Design For Verifiability" methodology for such parallel arrays.

In the case where the circuit under consideration consists of an interconnection of various components, this hierarchical aspect can be taken into account within PREVAIL™: several proof tools are integrated, and each component can be verified using the most appropriate system. The team of Flavio Wagner is developing a real framework-based environment for supporting hierarchical proof, library management, etc... [23].

REFERENCES

[1] P.ASHAR, A.GHOSH, S.DEVADAS, A.NEWTON : "Combinational and sequential logic verification using general binary decision diagrams". Proc. Int. Workshop on Logic Synthesis, Research Triangle Park (NC), May 1991.

[2] R.AUBIN : "Mechanizing structural induction-Part II : Strategies". Theoretical Computer Science 9. North-Holland,1979. pp. 347-362.

[3] R.S.BOYER, J S.MOORE : "A Computational Logic". ACM Monograph Series. Academic Press, Inc. 1979.

[4] R.S.BOYER, J S.MOORE : "A Computational Logic Handbook". Perspectives in Computing, Vol. 23. Academic Press, Inc. 1988.

[5] D.BORRIONE, L.PIERRE, A.SALEM : "Formal Verification of VHDL Descriptions in the PREVAIL Environment". IEEE Design&Test magazine, vol. 9, n°2, June 1992.
[6] R.E.BRYANT : "Graph-based Algorithms for Boolean Function Manipulation". IEEE Transactions on Computers, Vol. C-35, n°8, August 1986.
[7] R.E.BRYANT : "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication". IEEE Transactions on Computers, Vol. 40, n°2, February 1991.
[8] J.R.BURCH : "Using BDDs to verify multipliers". Proc. DAC'91, San Francisco (CA), June 1991.
[9] P.CAMURATI, P.PRINETTO : "Formal Verification of Hardware Correctness : Introduction and Survey of Current Research". IEEE Computer, Vol.21, n°7. July 1988.
[10] M.FUJITA, H.FUJISAWA, N.KAWATO : "Evaluation and Improvements of Boolean Comparison Method based on Binary Decision Diagrams". Proc. Int. Conference on Computer-Aided Design ICCAD'88, 1988.
[11] W.A.HUNT : "FM8501 : A verified microprocessor". Institute for Computing Science, University of Texas, Austin (USA). Technical Report n°47. February 1986.
[12] K.HWANG : "Computer arithmetic : principles, architecture and design", John Wiley & sons Inc., New-York, 1979.
[13] IEEE Standard VHDL Language Reference Manual, IEEE. 1988.
[14] T.KROPF : "Benchmark-Circuits for Hardware Verification, 2nd TPCD Conference". 2nd Conference on Theorem Proving in Circuit Design, Bad Herrenalb (Germany), 1994.
[15] J.S.MOORE : "Introducing Iteration into the Pure Lisp Theorem Prover". IEEE Transactions on Software Engineering, Vol. SE-1, n°3. September 1975.
[16] Z. MANNA, R. WALDINGER : "Knowledge and Reasoning in Program Synthesis". Artificial Intelligence Journal. Vol 6, 2. 1975.
[17] S.MALIK, A.R.WANG, R.K.BRAYTON, A.SANGIOVANNI-VINCENTELLI : "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment". Proc. Int. Conference on Computer-Aided Design ICCAD'88, 1988.
[18] L.PIERRE : "The Formal Proof of the Min-Max sequential benchmark described in CASCADE using the Boyer-Moore Theorem Prover". Proc. IFIP WG 10.2 Int. Workshop Nov. 1989. In "Formal VLSI Correctness Verification", L.Claesen ed., North Holland, 1990.
[19] L.PIERRE : "One Aspect of Mechanizing Formal Proof of Hardware : the Generalization of Partial Specifications". Proc. ACM International Workshop on Formal Methods in VLSI Design. Miami (Fl). 9-11 January 1991.
[20] L.PIERRE : "VHDL Description and Formal Verification of Systolic Multipliers". In "CHDL and their Applications", D.Agnew, L.Claesen & R.Camposano Eds, North Holland, 1993.
[21] H.SIMONIS : "Formal verification of multipliers". Proc. IFIP WG 10.2 Int. Workshop Nov. 1989. In "Formal VLSI Correctness Verification", L.Claesen ed., North Holland, 1990.
[22] D.VERKEST, L.CLAESEN, H.DE MAN : "Special Benchmark Session on Formal System Design". Proc. IFIP WG 10.2 Int. Workshop Nov. 1989. In "Formal VLSI Correctness Verification", L.Claesen ed., North Holland, 1990.
[23] F.WAGNER : "Prevail-DM : a framework-based environment for formal hardware verification". In "CHDL and their Applications", D.Agnew, L.Claesen & R.Camposano Eds, North Holland, 1993.

ACKNOWLEDGEMENTS

This work has been supported by the EEC under Charme-II ESPRIT Basic Research Working Group n°6018.

The author is grateful to the anonymous reviewers for their fruitful comments.