Describing and Verifying Synchronous Circuits with the Boyer-Moore Theorem Prover

Laurence PIERRE

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

Abstract. In this paper, we address the problem of finding a simple and efficient functional form for describing synchronous sequential circuits in the Boyer-Moore logic. By simple, we mean that it must be both user-readable and easily obtained by translation from a Hardware Description Language like VHDL. By efficient, we mean that it must be well-adapted to the proof mechanisms of the tool, Nqthm.
We propose two different recursive models, which are inspired from former results. We explain how they can be expressed in the Boyer-Moore logic, and we compare them on simple but illustrative examples. We also give the Nqthm proof of their equivalence. Finally, we conclude about their respective advantages and drawbacks.

I. INTRODUCTION.

To guarantee the correctness of circuits being designed, formal verification provides an alternative approach to simulation which is usually time-consuming and not always completely safe. In this framework, the circuit and its expected behaviour (called below the "specification") are described using a mathematical model, and formal reasoning is applied to this representation. Since we use the Boyer-Moore theorem prover Nqthm [BM,88], our mathematical model is functional. In our approach, formal proof means verifying that a given realization is equivalent to (or at least implies) its specification. Let us mention that the concept of formal proof covers various other aspects (special-purpose methods for verification of finite state machines, validation of temporal properties, ...), see the survey papers [CP,88], [Gu,92].

In cooperation with the team of D.Borrione, we are implementing a prototype proof environment, called PREVAIL [BP,92], to be embedded into a complete CAD system. This system takes as input circuit descriptions which are written in a synchronous subset of the Hardware Description Language VHDL [Ie,88], [BF,93]. The translation from VHDL to the input of the proof tool is automatic. Several proof tools (among them, Nqthm) are included in this environment, and the most appropriate one is selected depending on the features of the circuit to be validated.

Nqthm is essentially based on the principles of recursion and induction. Thus, it is well-adapted to categories of circuits that can naturally be expressed by recursive functions. In particular, we have evaluated its usefulness for describing and verifying replicated parameterized architectures [Pi,94], and synchronous sequential circuits :

- for repetitive devices, the recursive pattern translates the regularity of the structure,

- for sequential circuits, recursion implicitly represents the transformation of the circuit state between two time steps. This is the purpose of this paper.

We are interested in providing a functional model for synchronous sequential devices considered at the RT level, which satisfies the following criteria :

- this model must be acceptable by Nqthm, according to its definition principle,
- proof CPU times must be competitive with respect to other approaches,
- the automatic translation from a VHDL description of the circuit to this model must be feasible.

Related works.

For the purpose of simulation, synthesis or formal verification, various models have been proposed for the representation of sequential devices by means of recursive equations. Let us mention some of them :

- In [Jo,84/86], S.Johnson explains that functional notation is natural for the description of digital circuits and proposes a method to synthesize synchronous devices starting from recursive functional specifications. Successive steps of correctness-preserving transformations yield an iterative description that corresponds to an implementation, which is correct by construction.

- J.O'Donnell defines a method, called HDRE, for recursively modelling hardware [OD,87]. He uses stream recursion equations to describe the behaviour of a circuit. A sequential system is represented by one function of the primary inputs, the state variables are local variables which are updated by means of recursive equations.

- In [Br,89], A.Bronstein develops a "String-Functional Semantics" which is inspired from the notion of streams given in [Ka,74]. He associates an equation with each element in the circuit, which relates the output to the inputs. This system of equations is recursive as soon as the circuit includes structural loop(s).

These authors have probably been influenced by the results of M.Gordon. In [Go,80], he associates a sequential machine with a tuple $(S_M, out_M, next_M, s_M)$ where

S_M is the domain of states, $s_M \in S_M$ is the starting state,
$out_M : IN \times S_M \rightarrow OUT$ is the output function,
$next_M : IN \times S_M \rightarrow S_M$ is the next state function,

Then he gives the following "behaviour function" f_M, where B is the least solution of the domain equation $B = IN \rightarrow (OUT \times B)$:

$$f_M : S_M \rightarrow B$$
$$s \rightarrow f_M\, s = \lambda\, i\, .\, (out_M\, (i,s)\, ,\, f_M\, (next_M\, (i,s)))$$

On each recursive call, $out_M(i,s)$ computes the new output values, and $next_M(i,s)$ updates the values of the state variables. The behaviour of the machine is defined by $b_M = f_M\, s_M$. This representation is very close to the usual view of a Mealy machine, but the behaviour is modelled by a fix point equation.

In [Go,84], he refines this model and gives an implementation in the language LSM.

Synchronous circuits and Nqthm.

We will be strongly inspired by all these previous works, but let us recall that our aim is to determine a Nqthm-oriented model, and that this model must be such that we will be able to generate automatically the corresponding functions from our synchronous VHDL subset. In the following, we propose two recursive formulations and their Nqthm implementations. One of them is very close to the model above, the function takes as inputs the circuit primary inputs and state variables, and recursion is in fact pseudo-iteration. The other one characterizes each state variable and primary output by a higher-order function, and (mutual) recursion appears if the circuit contains structural loops. We give a mechanical proof of the equivalence of these models, using Nqthm.

We will see that the pseudo-iterative one is more interesting as far as Nqthm is concerned, from the point of view of efficiency as well as from the point of view of translation. This claim is illustrated by the Boyer-Moore verification of two simple examples : a BCD code recognizer [Di,78], and an implementation of the factorial function [Ca,87].

II. OVERVIEW OF NQTHM.

The Boyer-Moore system, Nqthm [BM,88], is based on a quantifier-free first order logic. Its main principles are :

- **The shell principle**, which is used to define inductive abstract data types by means of : a bottom object, a constructor, and one or more accessors. A boolean function, called a recognizer, checks whether an object belongs to the shell.
  Natural numbers is a well-known example of such an abstract data type : the bottom object of this type is **0**, the constructor is **+1**, and the accessor (the inverse of the function +1) is **-1**. The predicate recognizer of this type in Nqthm is called **numberp**.

- **The definition principle**, which allows to define recursive functions, with a strong verification of the correctness of the recursive form by the system. There must be a measure which decreases on each recursive call. For instance, we can define recursively the function "times" over natural numbers :

  $$times(i, j) =_{def} \textbf{if } i = 0 \textbf{ then } 0 \textbf{ else } j + times(i-1, j)$$

  The system finds out that the measure of the first parameter decreases on each recursive call, and that the recursion stops when i equals 0. Thus this definition is acceptable under the definition principle.

- **The induction principle**, on which the induction heuristics of the proof mechanism is based. An induction scheme is automatically generated according to the definition(s) of the recursive function(s) involved in the theorem to be proved. For example, if we want to verify the following proposition P(x,y) :

  $$numberp(x) \text{ and } numberp(y) \Rightarrow times(x, y+1) = x+times(x, y)$$

  The induction scheme generated for the proof of P(x,y) is :

  1. $x = 0 \Rightarrow P(x,y)$
  2. $(x \neq 0)$ and $P(x-1,y) \Rightarrow P(x,y)$

This prover can be applied to various fields : proof of mathematical theorems, validation of algorithms, hardware verification. Among the most important applications, we can mention the proof of the Church-Rosser theorem [Sh,85], of the Gauss law of quadratic reciprocity [Ru,92], of theorems in group theory [Yu,90], of the arithmetic-geometric mean theorem [KP,94], of the Goedel's incompleteness theorem, etc... and the verification of many algorithms such as a real-time control algorithm, compilers, the invertibilty of a public key encryption algorithm, etc...

As far as hardware verification is concerned, many applications have been developed, for instance :

- one of the first significant proofs has been achieved by W.Hunt who validated the FM8501 microprocessor [Hu,86], and also the ALU of the FM8502 [HB,89],
- the "String-Functional Semantics" mentioned above has been implemented in the Boyer-Moore logic, and devices such as pipelined architectures have been verified [BT,89],
- at IMEC (Belgium), Nqthm has been applied to the verification of parameterized architectures [VV,92]. Proving parameterized hardware with the Boyer-Moore system was also reported in [GW,85],
- at the University of Newcastle, this prover was included within a synthesis environment for DSP devices, to verify the correctness of such circuits during the synthesis process [BK,91].

In all these approaches, the Boyer-Moore code has been manually generated. Our purpose is mechanized proof and also *automatic translation.*

III. TWO POSSIBLE RECURSIVE FUNCTIONAL MODELS.

All along this paragraph, we consider a discrete time scale which corresponds to the rising edges of the main clock. The circuit is considered at the Register Transfer Level and is characterized by :

- a vector of primary inputs $< I_1, I_2, \ldots, I_n > = I$,
- a vector of primary outputs $< O_1, O_2, \ldots, O_m > = O$,
- a vector of state variables $< S_1, S_2, \ldots, S_q > = S$, and the initial state s_0,
- the (vectorial) state function $\varphi : \mathcal{I} \times \mathcal{S} \rightarrow \mathcal{S}$, where $\mathcal{I}$ is the input alphabet and $\mathcal{S}$ is the set of states,
- the (vectorial) output function $\psi : \mathcal{I} \times \mathcal{S} \rightarrow \mathcal{O}$, where $\mathcal{O}$ is the output alphabet.

III.1 First form.

The first representation associates a function S_j with each state variable (i.e. memory element) and a function O_j with each primary output. Primary inputs are functions of time, and state variables and primary outputs, which depend on these primary inputs, correspond to higher-order functions, i.e.

$$\forall\, j\,,\ 1 \leq j \leq n\,,\ I_j : \begin{array}{l} \mathbb{N} \rightarrow \mathcal{I}_j \\ t \rightarrow I_j(t) \end{array}$$

$$\forall\, j\,,\ 1 \leq j \leq q,\ S_j : (\mathbb{N} \rightarrow \mathcal{I}) \rightarrow (\mathbb{N} \rightarrow \mathcal{S}j)$$

$$\forall\, j\ ,\ 1 \le j \le m,\ O_j : (\mathbb{N} \to \mathfrak{I}) \to (\mathbb{N} \to \mathfrak{O}_j)$$
$$I \to (t \to O_j(I)(t))$$

Thus, the inputs, state variables and outputs can be globally represented by the functions :

$$I : \mathbb{N} \to \mathfrak{I} = \prod_{j=1}^{n} \mathfrak{I}_j$$
$$t \to I(t) = < I_1(t), I_2(t), \ldots, I_n(t) >$$

$$S : (\mathbb{N} \to \mathfrak{I}) \to (\mathbb{N} \to \mathfrak{S})$$
$$I \to (t \to S(I)(t) = < S_1(I)(t), S_2(I)(t), \ldots, S_q(I)(t) >)$$

where $\mathfrak{S}$ is the product $\prod_{j=1}^{q} \mathfrak{S}_j$ of the sets of values of the functions $S_1, \ldots, S_q$.

$$O : (\mathbb{N} \to \mathfrak{I}) \to (\mathbb{N} \to \mathfrak{O})$$
$$I \to (t \to O(I)(t) = < O_1(I)(t), O_2(I)(t), \ldots, O_m(I)(t) >)$$

where $\mathfrak{O}$ is the product $\prod_{j=1}^{m} \mathfrak{O}_j$.

Let *C* denote the function that describes the whole synchronous sequential circuit, for all $t \ge 0$, C(I)(t) consists in the pair <S(I)(t), O(I)(t)>, i.e. we have :

$$C(I)(t) = < S(I)(t), O(I)(t) >$$

Let us see what are the definitions of the functions S and O. The registers are memory elements with a one-unit delay, thus they depend on the past values of the primary inputs and also on the past values of themselves if there are structural loops. The outputs depend on the current values of the primary inputs and of the state variables. It means that S and O are associated with the following higher-order functions :

$$S(I)(t) = \text{if } (t = 0) \text{ then } s_0 \text{ else } \varphi(I(t\text{-}1), S(I)(t\text{-}1))$$
$$O(I)(t) = \psi(I(t), S(I)(t))$$

An advantage of this representation is that it clearly expresses the circuit structure. Moreover, automatic translation from a HDL description to this formalism could be feasible. However, Nqthm does not support higher-order logic and this model must significantly be modified to fit the Boyer-Moore logic. For that reason, even if this formulation can be implemented in Nqthm, the examples will show that this is not the best choice. It could probably be more interesting in the framework of a higher-order proof assistant such as HOL [Go,85].

But let us examine how this model can be transformed to be implemented in Nqthm. First, since we have to express it in first-order logic, we use the method which consists in considering the history of the inputs. Instead of describing primary inputs as time functions, we consider them as lists of values which represent their histories. Thus,

output and state variable functions are simply first-order functions. For lists of bits, we use the bit-vector shell given in [Hu,86], it is characterized by :

- the bottom : *(btm)*
- the constructor : *bitv*
- the accessors : *bit* (first bit) and *vec* (rest of the vector)
- the recognizer : *bitvp*

Another problem with Nqthm is that our model involves mutually recursive functions, and Nqthm does not support mutual recursion. To avoid this problem, we use a well-known trick : we write a unique function, with an extra parameter (a "flag"), which contains all bodies of the mutually recursive forms. The appropriate body part is selected according to the flag value.

Example. We consider a device which checks if a four-bit sequence is a valid BCD code. Two possible implementations are proposed in [Di,78], and we will verify their equivalence in the next paragraph. First, let us just describe one of these circuits, given by Figure 1 below, with the model above.
I is the input, O is the output, and S_1, S_2, S_3 and S_4 are registers. The four input bits are analyzed sequentially, starting from the least significant one. The output is significant when the four bits have been read, i.e. after four time units. The registers S_3 and S_4 must be initialized to *false*, the initial values of S_1 and S_2 have no importance, we choose to give each of them the initial value *false*.

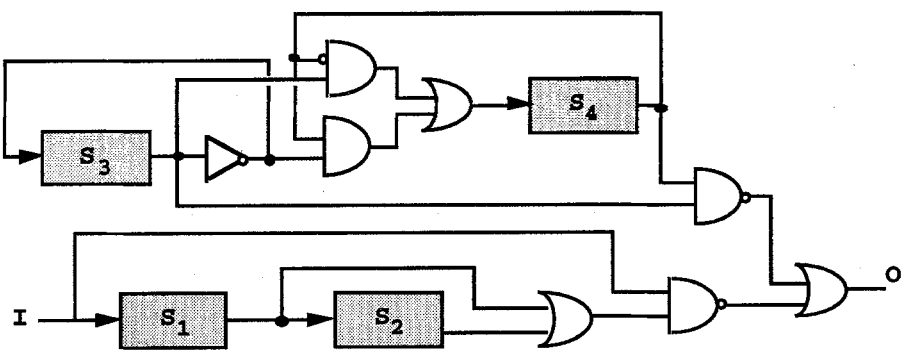


Figure 1 : Simple implementation of the BCD code recognizer

At any time t equal to 3 modulo 4, the sequence <I(t), I(t-1), I(t-2), I(t-3)> represents a valid BCD code (i.e. a natural number<10) if the following boolean expression holds :

not I (t) or (not I (t -1) and not I (t -2))

At the same time, the value of the output O is *true* in that case, and *false* otherwise.

This circuit will be referred to as BCD_1. The function C_{BCD1} for this device is :

$$C_{BCD1}(I)(t) = < < S_1(I)(t), S_2(I)(t), S_3(I)(t), S_4(I)(t) >, O(I)(t) >$$

with

$S_1(I)(t)$ =def *if t = 0 then* ***false*** *else I (t-1)*

$S_2(I)(t)$ =def *if t = 0 then* ***false*** *else* $S_1(I)(t-1)$

$S_3(I)(t)$ =def *if t = 0 then* ***false*** *else* ***not*** $S_3(I)(t-1)$

$S_4(I)(t)$ =def *if t = 0*
then ***false***
else (***not*** $S_4(I)(t-1)$ ***and*** $S_3(I)(t-1)$) ***or*** ($S_4(I)(t-1)$ ***and not*** $S_3(I)(t-1)$)

$O(I)(t)$ =*def* ***not*** ($S_4(I)(t)$ ***and*** $S_3(I)(t)$) ***or not*** (*I (t)* ***and*** ($S_1(I)(t)$ ***or*** $S_2(I)(t)$))

When we translate these functions in Nqthm, the input function I is represented by the list of its successive values, i.e. a bit sequence denoted i, and i = <I(t), I(t-1), ..., I(0)>. The function definitions, given below in the Boyer-Moore syntax, check that their parameter i satisfies the recognizer "bitvp". The function `S-BCD1` is the global expression of S_1, S_2, S_3, and S_4, and `O-BCD1` corresponds to O. We can remark that, in this particular case, there is no mutual recursion and that we could have defined S_1, S_2, S_3, and S_4 as four independent functions. We have preferred the general methodology, in particular to guarantee a similar processing of this circuit and of its alternative implementation that will be seen in the next paragraph.

```
(defn S-BCD1 (flag i)
  (if (bitvp i)
      (if (equal flag 's1)
          (if (equal i (btm))
              f
              (if (equal (vec i) (btm))
                  f     ; initial value of s1
                  (bit (vec i))))
          (if (equal flag 's2)
              (if (equal i (btm))
                  f
                  (if (equal (vec i) (btm))
                      f     ; initial value of s2
                      (S-BCD1 's1 (vec i))))
              (if (equal flag 's3)
                  (if (equal i (btm)) f
                      (if (equal (vec i) (btm))
                          f     ; initial value of s3
                          (not (S-BCD1 's3 (vec i)))))
                  (if (equal flag 's4)
                      (if (equal i (btm)) f
                          (if (equal (vec i) (btm))
                              f     ; initial value of s4
                              (or (and (not (S-BCD1 's4 (vec i)))
                                       (S-BCD1 's3 (vec i)))
                                  (and (S-BCD1 's4 (vec i))
                                       (not (S-BCD1 's3 (vec i))))))))
                      f))))
      f))

(defn O-BCD1 (i)
  (if (bitvp i)
      (if (equal i (btm))
          f
          (or (not (and (S-BCD1 's4 i) (S-BCD1 's3 i)))
              (not (and (bit i) (or (S-BCD1 's1 i) (S-BCD1 's2 i))))))
      f))
```

III.2 Second form.

The second format is closer to the model proposed by M.Gordon. It consists in a unique first-order tail-recursive (i.e. pseudo-iterative) function, referred to as *C'*, which expresses the behaviour of the circuit from time 0 to the time point given by the length of i.

$$C' : \bigcup_{k=1}^{\infty} \mathfrak{I}^k \times \mathfrak{S} \rightarrow \mathfrak{S} \times \mathfrak{O}$$

$$(i,\ s) \rightarrow \textit{if length}\ (i) = 1$$
$$\textit{then} < s,\ \psi(\textit{head}(i), s) >$$
$$\textit{else}\ C'(\textit{tail}\ (i),\ \varphi(\textit{head}(i),\ s))$$

The parameters of this function are the (vector of) inputs, and the (vector of) state variables. The inputs are represented by their histories, and the functions denoted *head*, *tail* and *length* give respectively the first element of the sequence, the sequence without its first element, and the length of the sequence; here i = <I(0), ..., I(t-1), I(t)>. State variables are accumulating parameters that are updated on each recursive call.

Example. For the circuit of Figure 1, the function C'_{BCD1} associated with the second model is :

$C'_{BCD1}\ (i, < s_1, s_2, s_3, s_4 >)$ =def
if length (i) = 1
then $< s_1, s_2, s_3, s_4,$ ***not*** $(s_4$ ***and*** $s_3)$ ***or not*** *(head (i)* ***and*** $(s_1$ ***or*** $s_2))$ >
else C'_{BCD1} (*tail (i),*
$<$ *head (i),* s_1 , ***not*** s_3,
(not s_4 ***and*** $s_3)$ ***or*** $(s_4$ ***and not*** $s_3)$ $>$)

and it is true that

$$C'_{BCD1}\ (< I\ (0),\ I\ (1),\ \ldots,\ I\ (t) >,\ < \textbf{\textit{false, false, false, false}} >) = C_{BCD1}\ (I)(t)$$

In the corresponding Nqthm representation, the condition "length(i) = 1" is replaced by "tail(i) is empty" for reasons of efficiency. In fact, since i is a bit-vector in this example, "head" corresponds to `bit`, "tail" corresponds to `vec`, and thus "tail(i) is empty" is expressed by "vec(i) = (btm)". Therefore, the Boyer-Moore function associated with C'_{BCD1} is :

```
(defn C2-BCD1 (i s1 s2 s3 s4)
  (if (bitvp i)
      (if (equal i (btm))
          f
          (if (equal (vec i) (btm))
              (or (not (and s4 s3)) (not (and (bit i) (or s1 s2))))
              (C2-BCD1 (vec i) (bit i) s1 (not s3)
                       (or (and (not s4) s3) (and s4 (not s3))))))
      f))
```

To be acceptable under the definition principle, this function also has to check that its parameter i satisfies the recognizer "bitvp". Another remark is that, since we are not interested in the final register values, this function only returns the final output value.

III.3 Equivalence of these two models.

With Solange Coupet-Grimal, we have made a hand-proof of the equivalence of these two models [CP,91], i.e. we have :

$$\forall (t, I) \in \mathbb{N} \times (\mathbb{N} \rightarrow \mathcal{I}), \quad C(I)(t) = C'(< I(0), I(1), \ldots, I(t) >, s_0)$$

In fact, there are at least two ways of achieving this proof. First, we can use general results about transformation of recursion into pseudo-iteration (see for instance [BD,77], [HL,78], [HK,92]). Second, a simpler inductive proof is also feasible since the problem here is less general than the ones considered in the previous references. We have proposed both hand-proofs.

Here, we give a mechanical proof of this equivalence, using Nqthm. To fit the Boyer-Moore logic, the first model has been modified such that I is considered as the history of its successive values. Thus, the history is seen as < I(t), I(t-1), ... , I(1), I(0) > in the first representation (where we reason from the current time to the initial one), and is considered as < I(0), I(1), ... , I(t-1), I(t) > in the second one (where we "look at" the future). In the following proof script, we first define the "shell" of sequences of anything (with recognizer `sequp`), and various functions about these sequences, in particular `revseq` which allows one to reverse a sequence, to be able to deal with <I(t), I(t-1), ..., I(0)> and < I(0), ... , I(t-1), I(t) > in the same theorems.

```
; Declaration of the shell of sequences (of anything) :
(add-shell sequ     ; constructor
           empty    ; bottom
           sequp    ; recognizer
           ((first (none-of) false)       ; first element.
            (rest (one-of sequp) empty))) ; rest of the sequence.

(dcl phi (i s))     ; We don't need to know the definitions of φ, ψ
(dcl psi (i s))     ; and error.
(dcl error ())
; ("dcl" allows one to declare a function identifier without giving
;  the function body)

; Functions on sequences :
(defn appendseq (v1 v2)    ; concatenation of 2 sequences
  (if (sequp v1)
      (if (equal v1 (empty))
          v2
          (sequ (first v1) (appendseq (rest v1) v2)))
      v2))

(defn sizeseq (x)     ; size of a sequence
  (if (sequp x)
      (if (equal x (empty)) 0 (add1 (sizeseq (rest x))))
      0))
```

```
(defn revseq (v)     ; reverse of a sequence
   (if (sequp v)
       (if (equal v (empty))
           (empty)
           (appendseq (revseq (rest v)) (sequ (first v) (empty))))
       (empty)))

(defn lastseq (s)     ; last element of a sequence
  (if (sequp s)
      (if (equal s (empty))
          f
          (if (equal (rest s) (empty)) (first s) (lastseq (rest s))))
      f))

; + some elementary lemmas on these functions.

; Description of Model 1 (the model of § III.1) :

(defn S (seqi time state0)  ; here, seqi = < I(t), I(t-1), ... I(0) >
   (if (numberp time)
       (if (equal time 0)
           state0
           (phi (first (rest seqi))
                (S (rest seqi) (sub1 time) state0)))
       (error)))

(defn O (seqi time state0)  ; here, seqi = < I(t), I(t-1), ... I(0) >
   (psi (first seqi)
        (S seqi time state0)))

; To facilitate the proof, we introduce the intermediate function S' :
(defn Sprime (seqi time state0)
   (if (numberp time)
       (if (equal time 0)
           state0
           (phi (first seqi)
                (Sprime (rest seqi) (sub1 time) state0)))
       (error)))

; and we prove the equivalence between the terms S(x.v, time, s0) and
; S'(v, time, s0) :
(prove-lemma equiv-S-Sprime (rewrite)
  (equal (S (sequ x v) time s0)
         (Sprime v time s0)))

; Description of Model 2 (the model of § III.2) :

(defn C (seqi state)     ; here, seqi = < I(0), …, I(t-1), I(t) >
   (if (sequp seqi)
       (if (equal seqi (empty))
           (error)
           (if (equal (rest seqi) (empty))
               (list state (psi (first seqi) state))
               (C (rest seqi) (phi (first seqi) state))))
       (error)))

; To facilitate the proof, we introduce the intermediate function S2
; (which models C without the output) :
```

```
(defn S2 (seqi state)
   (if (sequp seqi)
       (if (equal seqi (empty))
           (error)
           (if (equal (rest seqi) (empty))
               state
               (S2 (rest seqi) (phi (first seqi) state))))
       (error)))

; ... and the simpler form S2' :
(defn S2prime (seqi state)
   (if (sequp seqi)
       (if (equal seqi (empty))
           state
           (S2prime (rest seqi) (phi (first seqi) state)))
       (error)))

; and we prove the equivalence between the terms S2(v.x.empty, s0) and
; S2'(v, s0) :
(prove-lemma equiv-S2-S2prime (rewrite)
   (implies (sequp v)
            (equal (S2 (appendseq v (sequ x (empty))) s0)
                   (S2prime v s0))))

; Proof of the equivalence :

; ----- 1. Equivalence between S2' and S' ----- :

(prove-lemma generalization-for-equiv-Sprime-S2prime (rewrite)
   (implies (sequp v)
            (equal (phi a (S2prime v state))
                   (S2prime (appendseq v (sequ a (empty))) state))))

(prove-lemma equiv-Sprime-S2prime (rewrite)
   (implies (and (sequp seqi) (numberp time)
                 (equal (sizeseq seqi) time))
            (equal (Sprime seqi time state0)
                   (S2prime (revseq seqi) state0))))
; S'(seqi, time, s0) = S2'(revseq(seqi), s0)

; ----- 2. Equivalence between S2 and S ----- :

(prove-lemma equiv-S-S2 (rewrite)
   (implies (and (sequp seqi) (numberp time)
                 (equal (sizeseq seqi) (add1 time)))
            (equal (S seqi time state0) (S2 (revseq seqi) state0))))
; S(seqi, time, s0) = S2(revseq(seqi), s0)

; ----- 3. Final equivalence ----- :

(prove-lemma equiv-C-list-S2-psi (rewrite)
   (implies (and (sequp seqi) (not (equal seqi (empty))))
            (equal (C seqi s)
                   (list (S2 seqi s)
                         (psi (lastseq seqi) (S2 seqi s))))))
; C(seqi, s) = < S2(seqi, s), ψ(lastseq(seqi), S2(seqi, s)) >
```

```
(prove-lemma equiv-C-list-S2-psi-bis (rewrite)
   (implies (and (sequp seqi) (not (equal seqi (empty))))
            (equal (C (revseq seqi) s)
                   (list (S2 (revseq seqi) s)
                         (psi (first seqi) (S2 (revseq seqi) s))))))
   ((use (equiv-C-list-S2-psi (seqi (revseq seqi))))))  ; hint
; C(revseq(seqi), s) = < S2(revseq(seqi), s),
;                        ψ(first(seqi), S2(revseq(seqi), s)) >

(prove-lemma equiv-C-list-S-psi (rewrite)
   (implies (and (sequp seqi) (numberp time)
                 (equal (sizeseq seqi) (add1 time)))
            (equal (C (revseq seqi) s)
                   (list (S seqi time s)
                         (psi (first seqi) (S seqi time s)))))
   ((use (equiv-C-list-S2-psi-bis) (equiv-S-S2 (state0 s)))))  ; hint
; C(revseq(seqi), s) = < S(seqi, time, s),
;                        ψ(first(seqi), S(seqi, time, s)) >

(prove-lemma equivalence ()
   (implies (and (sequp seqi) (numberp time)
                 (equal (sizeseq seqi) (add1 time)))
            (equal (C (revseq seqi) state0)
                   (list (S seqi time state0) (O seqi time state0)))))
; C(<I(0), …,I(t-1),I(t)>, s0) = < S(<I(t), I(t-1), …, I(0)>, t, s0),
;                                  O(<I(t), I(t-1), …, I(0)>, t, s0) >
```

The use of a theorem prover gives a higher level of confidence w.r.t. the correctness of this proof than a hand-proof can give. Human errors can be made when encoding the original problem, but the prover might usually detect them. In fact, during this mechanical verification, we discovered a small error in the manual verification (we assumed a wrong intermediate result, but the way we used it made the proof succeed. Nqthm helped us discovering that this result was wrong). Moreover, in the hand-proof, we used one more intermediate representation and associated extra lemmas, and the mechanical proof revealed the uselessness of this intermediate function. In conclusion, this Nqthm proof is more safe and elegant than the former manual proof. The total CPU time is about 7 seconds on a SUN SPARCclassic.

IV. PRACTICAL COMPARISON WITH THE BCD CODE RECOGNIZER.

For this practical comparison, we consider a second possible implementation of the BCD code recognizer, also taken in [Di,78], and given by Figure 2 below. This circuit, that will be referred to as BCD_2, is an optimization of the first one, in the sense that there are only 3 registers S_1, S_2 and S_3, that have to be initialized to false.

IV.1 First representation.

The function C_{BCD2} which corresponds to the first model of this device is :

$$C_{BCD2}(I)(t) = < < S_1(I)(t), S_2(I)(t), S_3(I)(t) >, O(I)(t) >$$

with

$S_1(I)(t) =_{def}$ *if t = 0 then* ***false***
else ***(not*** *I (t-1)* ***and*** $S_1(I)(t-1)$ ***and*** $S_3(I)(t-1)$***)***
or (not *I (t-1)* ***and*** $S_2(I)(t-1)$ ***and not*** $S_3(I)(t-1)$***)***

$S_2(I)(t)$ =def *if* $t = 0$
then ***false***
else (***not*** $S_1(I)(t\text{-}1)$ ***and not*** $S_2(I)(t\text{-}1)$ ***and not*** $S_3(I)(t\text{-}1)$)
or ($I(t\text{-}1)$ ***and not*** $S_1(I)(t\text{-}1)$ ***and not*** $S_3(I)(t\text{-}1)$)

$S_3(I)(t)$ =def *if* $t = 0$
then ***false***
else $S_2(I)(t\text{-}1)$ ***or*** ($I(t\text{-}1)$ ***and*** $S_1(I)(t\text{-}1)$ ***and*** $S_3(I)(t\text{-}1)$)

$O(I)(t)$ =def ***not*** ($I(t)$ ***and not*** $S_1(I)(t)$ ***and not*** $S_2(I)(t)$ ***and*** $S_3(I)(t)$)

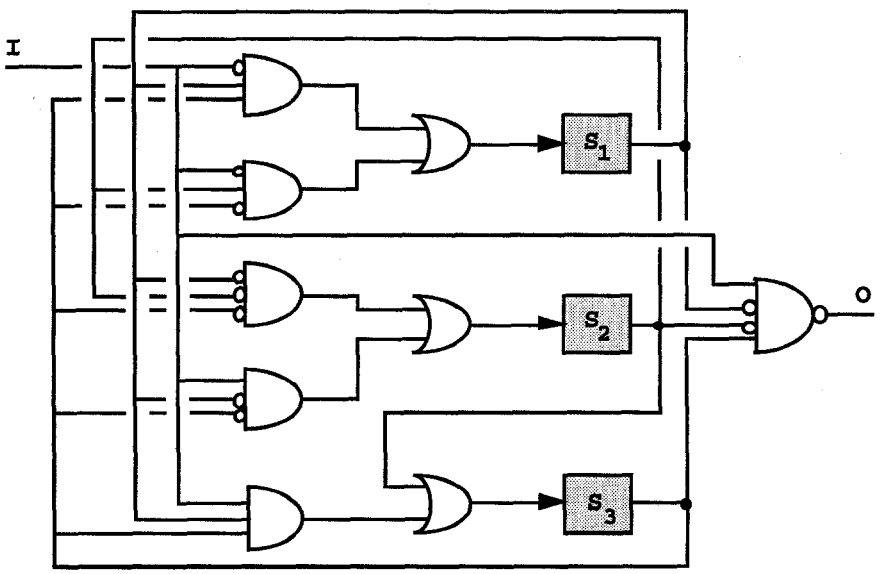


Figure 2 : Second implementation of the BCD code recognizer

In the Nqthm format, the input function I(t) is represented by the list of its successive values, i.e. a bit sequence denoted i. The function S-BCD2 is the global expression of S_1, S_2, and S_3, and O-BCD2 is associated with O :

```
(defn O-BCD2 (i)
  (if (bitvp i)
      (if (equal i (btm))
          f
          (not (and (bit i) (not (S-BCD2 's1 i))
                    (not (S-BCD2 's2 i)) (S-BCD2 's3 i))))
      f))

(defn S-BCD2 (flag i)
  (if (bitvp i)
      (if (equal flag 's1)
          (if (equal i (btm))
              f
              (if (equal (vec i) (btm))
                  f   ; initial value of s1
```

```
                (or (and (not (bit (vec i))) (S-BCD2 's1 (vec i))
                         (S-BCD2 's3 (vec i)))
                    (and (not (bit (vec i))) (S-BCD2 's2 (vec i))
                         (not (S-BCD2 's3 (vec i)))))))
                (if (equal flag 's2)
                    ; ....  similarly for S2
                    (if (equal flag 's3)
                        ; ....  similarly for S3
                        f)))
    f))
```

IV.2 Second representation.

The function C'_{BCD2} below expresses this circuit according to the second model :

$C'_{BCD2}(i, < s_1, s_2, s_3 >)$ =def
if length (i) = 1
then $< s_1, s_2, s_3,$ ***not*** *(head (i)* ***and not*** s_1 ***and not*** s_2 ***and*** $s_3) >$
else C'_{BCD2} *(tail (i),*
$<$ ***(not*** *head (i)* ***and*** s_1 ***and*** $s_3)$ ***or***
(not *head (i)* ***and*** s_2 ***and not*** $s_3)$,
(not s_1 ***and not*** s_2 ***and not*** $s_3)$ ***or***
(head (i)*** and not*** s_1 ***and not*** $s_3)$,
s_2 ***or (***head (i)*** and*** s_1 ***and*** $s_3) >$)

and it is true that

$$C'_{BCD2}(< I(0), I(1), ..., I(t) >, < \textbf{false}, \textbf{false}, \textbf{false} >) = C_{BCD2}(I)(t)$$

Its Nqthm formulation is the function `C2-BCD2`, where "length(i) = 1" is translated by the condition `(equal (vec i) (btm))`. Like in the case of the first implementation of this BCD code recognizer, this function only returns the value of the output O.

```
(defn C2-BCD2 (i s1 s2 s3)
  (if (bitvp i)
      (if (equal i (btm))
          f
          (if (equal (vec i) (btm))
              (not (and (bit i) (not s1) (not s2) s3))
              (C2-BCD2 (vec i)
                       (or (and (not (bit i)) s1 s3)
                           (and (not (bit i)) s2 (not s3)))
                       (or (and (not s1) (not s2) (not s3))
                           (and (bit i) (not s1) (not s3)))
                       (or s2 (and (bit i) s1 s3)))))
      f))
```

IV.3 Verification process.

Our goal is the verification of the equivalence of the implementations of Figures 1 and 2. In other words, we prove the functional equivalence of `O-BCD1` and `O-BCD2` on the one hand, and of `C2-BCD1` and `C2-BCD2` on the other hand. Of course, this is an academic example that has been chosen here for its simplicity, but it is clear that a

specialized FSM equivalence checker should be more efficient than Nqthm on this particular benchmark.

Each of the proposed circuits analyzes a four-bit sequence, outputs a significant result, and must be ready to analyze the next four-bit sequence. It means that the circuit must re-initialize its registers to the correct initial values at the right time. Therefore, our proof process also includes the verification of this initialization.

• First, the theorem `equivalence-of-BCD1-BCD2` below states that the functions `O-BCD1` and `O-BCD2` give the same result, for any 4-bit sequence `i` :

```
(prove-lemma equivalence-of-BCD1-BCD2 (rewrite)
   (implies (and (bitvp i) (equal (size i) 4))
            (equal (O-BCD1 i) (O-BCD2 i))))
```

The presence of the "flags" (i.e. enumerated values) in the function definitions, implies that Nqthm generates a lot of sub-cases, and thus we have a CPU time of 250 seconds for this proof (on a SUN SPARCstation 2).

Then, to verify that the re-initializations are correctly done before each analysis cycle, we prove that the registers S_3 and S_4 of BCD_1, and the registers of BCD_2 become *false* after 5 time units. For instance, for the register S_3 of the first circuit :

```
(prove-lemma init-s3-BCD1 (rewrite)
  (implies (and (bitvp i) (equal (size i) 5))
           (equal (S-BCD1 's3 i) f)))
```

The total CPU time for the initialization verification (five lemmas) is about 160 seconds on a SUN SPARCstation 2.

• Now, let us apply the same procedure to the second model. The following lemma verifies the equivalence of `C2-BCD1` and `C2-BCD2`, provided that states variables are correctly initialized :

```
(prove-lemma equivalence-of-C2-BCD1-BCD2 ()
  (implies (and (bitvp i) (equal (size i) 4)
                (boolp s1)       ; s1 and s2 of BCD1 can take any
                (boolp s2))      ; boolean values
           (equal (C2-BCD1 i s1 s2 f f) (C2-BCD2 i f f f)))))
```

In that case, we do not deal with enumerated types, and the proof only takes 22 seconds, on a SUN SPARC station 2.

Now, we verify that re-initializations are correctly done before the next cycle. For instance, the correct initialization of the register S_3 of BCD_1 is verified by the following theorem :

```
(prove-lemma init-s3-BCD1 ()
  (implies (and (bitvp i) (boolp s1) (boolp s2)
                (equal (size i) 5))                ; after 5 time units
           (equal (C2-BCD1-s3 i s1 s2 f f) f)))    ; s3 equals false
```

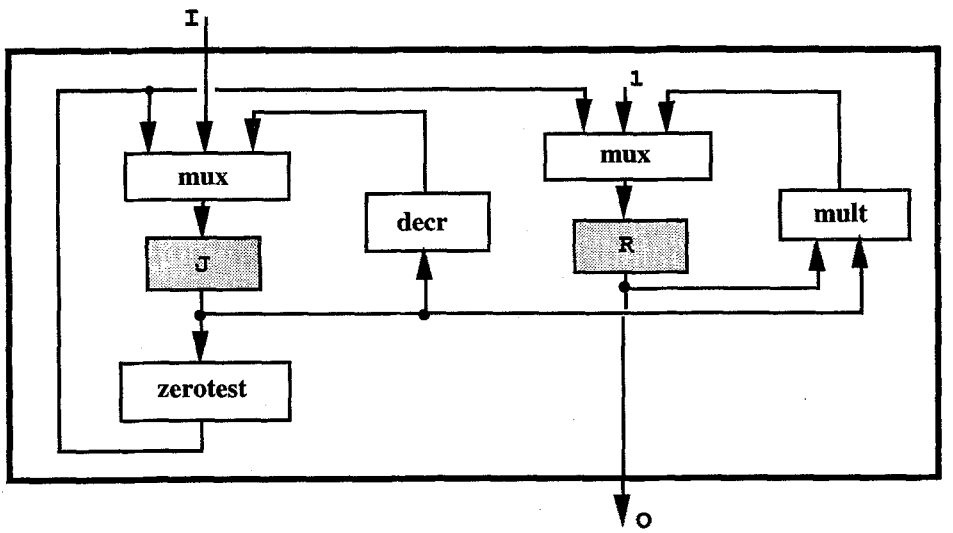
where the function `C2-BCD1-s3` is built on the same pattern as `C2-BCD1` but only returns the final value of S_3.

The re-initializations of the other registers are validated accordingly. The total CPU time for these verifications (five theorems) is 70 seconds on a SUN SPARCstation 2.

In both cases, the proofs are completely automatic, without intermediate lemmas. We can already draw a conclusion from this example : the difference between CPU times can be significant; this is due in particular to the flags of the first form which represent enumerated values and which make the system examine every possible combination.

V. PRACTICAL COMPARISON WITH THE FACTORIAL CIRCUIT.

Now, we propose a comparison using another kind of benchmark. We will reason at the arithmetic level, and we will also see the generalization problem that may arise in that case. Figure 3 depicts an implementation of a synchronous sequential device which iteratively computes the factorial of an integer I [Ca,87]. The duration of the computation cycle is I time units, and then O holds the result.



where : mux(c,i1,i2,o) ↔ o = if c then i1 else i2, decr(i,o) ↔ o = i-1
zerotest(i,o) ↔ o = (i=0), mult(i1,i2,o) ↔ o = i1*i2

Figure 3 : Implementation of the factorial function

We can verify that this circuit really computes I!, provided that the registers J and R are initialized respectively to 0 and 1. As a high-level specification, we use the following function that implements the simplest version of the factorial algorithm :

factorial (i) $=_{def}$ *if i=0 then 1 else i * factorial (i-1)*

V.1 Circuit descriptions.

The function C_{fact} which expresses the structure of this device according to the first model is :

$$C_{fact}(I)(t) = < <J\,(I)(t)\,,\, R\,(I)(t)>,\, O\,(I)(t) >$$

with

$J(I)(t) =_{def}$ *if t = 0 then 0 else (if J(I)(t-1) = 0 then I(t-1) else J(I)(t-1) -1)*

$R(I)(t) =_{def}$ *if t = 0 then 1 else (if J(I)(t-1) = 0 then 1 else R(I)(t-1) * J(I)(t-1))*

$O(I)(t) =_{def} R(I)(t)$

Since it has been possible to prove, before the verification of this circuit, that bit-level realizations of the components Mux, Mult, Decr and Zerotest respectively implement the arithmetic-level functions If, *, -1 and =0, we replace them by these functions.

In fact, we can remark that the value of the input I can be unchanged during a computation cycle, since it is not significant. Its next value is given before the next cycle (when the registers are re-initialized), for the next computation. Thus, there is no need to introduce the input history, and we can simplify the model by considering a simple variable i, and the time t. The representation above then becomes :

$$C_{fact}(i, t) = << J(i, t), R(i, t) >, O(i, t) >$$

with

$J(i, t) =_{def}$ *if t = 0 then 0 else (if J(i, t-1) = 0 then i else J(i, t-1) -1)*

$R(i, t) =_{def}$ *if t = 0 then 1 else (if J(i, t-1) = 0 then 1 else R(i, t-1) * J(i, t-1))*

$O(i, t) =_{def} R(i, t)$

Considering this remark and using the second model, we get the following function :

$C'_{fact}(i, t, < j, r >) =_{def}$
if t = 0
then $< < j, r >, r >$
else $C'_{fact}(i, t-1, <$ *if j = 0 then i else j -1, if j = 0 then 1 else r * j* $>)$

and we have

$$C'_{fact}(i, t, < 0, 1 >) = C_{fact}(i, t)$$

For the first representation, the Nqthm implementation is composed of two functions `S-fact` and `O-fact`, where `S-fact` is the global expression of J and R and `O-fact` is associated with O.

```
(defn S-fact (flag i time)
  (if (numberp i)
      (if (numberp time)
          (if (equal flag 'j)
              (if (equal time 0)
                  0
                  (if (equal (S-fact 'j i (sub1 time)) 0)
                      i
                      (sub1 (S-fact 'j i (sub1 time )))))
            (if (equal flag 'r)
                (if (equal time 0) 1
                    (if (equal (S-fact 'j i (sub1 time)) 0) 1
                        (times (S-fact 'r i (sub1 time))
                               (S-fact 'j i (sub1 time)))))
                0))
          0)
      0))
```

```
(defn O-fact (i time)
  (if (numberp i)
      (if (numberp time) (S-fact 'r i time) 0)
      0))
```

And as far as the tail-recursive model is concerned, the Boyer-Moore implementation corresponds to the function `C2-fact` below, which only returns the final value of O (that is the only interesting one with respect to the "observational" validation of the circuit).

```
(defn C2-fact (i time j r)
  (if (zerop time)
      r
      (C2-fact i (sub1 time) (if (equal j 0) i (sub1 j))
               (if (equal j 0) 1 (times r j)))))
```

V.2 Verification process.

We want to verify the correctness of this device with regard to its high-level specification, that has the following Nqthm interpretation :

```
(defn factorial (i)
  (if (zerop i) 1 (times i (factorial (sub1 i)))))
```

More precisely, we can prove that `O-fact` and `factorial` on the one hand, and `C2-fact` and `factorial` on the other hand give the same result after `i` time units. These statements correspond to the theorems `correctness-of-O-fact` and `correctness-of-C2-fact` below :

```
(prove-lemma correctness-of-O-fact ()
  (implies (numberp i)
           (equal (O-fact i i)
                  (factorial i))))

(prove-lemma correctness-of-C2-fact ()
  (implies (numberp i)
           (equal (C2-fact i i 0 1)
                  (factorial i)))))
```

Unfortunately, neither the first lemma nor the second one can be proved by induction unless we generalize it. This is due to the presence of the initial values 0 and 1 for the registers. This aspect will demonstrate another advantage of the second model : generalization is much more easy with that model.

Before achieving the proof of `correctness-of-O-fact`, the three intermediate lemmas below must be verified :

```
; j(i,t)= if t=0 then 0 else (i-t)+1   provided that t ≤ i :
(prove-lemma generalization-of-j (rewrite)
  (implies (and (numberp i) (numberp time) (leq time i))
           (equal (S-fact 'j i time)
                  (if (equal time 0)
                      0
                      (add1 (difference i time))))))
```

```
; ((i-t)+1) * r(i,t) = i * r(i-1,t)   provided that t≠0, i≠0 and t≤i :
(prove-lemma generalization-of-r-1 (rewrite)
  (implies (and (numberp i) (numberp time)
                (not (equal time 0)) (not (equal i 0)) (leq time i))
           (equal (times (add1 (difference i time))
                         (S-fact 'r i time))
                  (times i (S-fact 'r (sub1 i) time )))))

; 2 * r(t+1,t) = (t+1) * r(t,t)   provided that t≠0 :
(prove-lemma generalization-of-r-2 (rewrite)
  (implies (and (numberp time) (not (equal time 0)))
           (equal (times 2 (S-fact 'r  (add1 time) time))
                  (times (add1 time) (S-fact 'r time time))))
  ((use (generalization-of-r-1 (time time) (i (add1 time))))))
```

As for the proof of `correctness-of-C2-fact`, the theorem `generalization-for-C2-fact`, which generalizes both `j` and `r`, suffices :

```
(prove-lemma generalization-for-C2-fact (rewrite)
  (implies (and (numberp i) (numberp j) (numberp x))
           (equal (C2-fact x j (add1 j) i)
                  (times i (times (add1 j) (factorial j))))))
```

With this simple and short example, we can see that the generalization problem is of valuable help for comparing the approaches. We can imagine what could be, for a larger device, the difficulty of producing the intermediate theorems associated with the first model. Here, the analysis of the proof tracing was sufficient to find out the reasons of the proof failure and to deduce the appropriate lemmas, but it took several man hours. This approach is not feasible with a long and complicated proof. Moreover, this technique cannot be mechanized, even for small circuits.

Conversely, this mechanization is feasible with the tail-recursive form. We have proposed an automatic generalization algorithm for iterative arithmetic circuits described using this second model [Pi,91a]. Thus, there is no need for designers to have a deep understanding of the translation and proof processes, since they only have a minor (questions/answers) interaction with the system. The recursive function which models the implementation is automatically produced from the VHDL description, and the generalization tool generates the appropriate lemma.

VI. CONCLUSION.

We have defined two functional models for representing synchronous devices in the Boyer-Moore logic. We have also given a mechanical proof of their equivalence. The purpose of this study was to determine the most appropriate one for formal reasoning with Nqthm. From a practical point of view, the following conclusions can be drawn :

- we could have foreseen that the first model will not be very satisfying w.r.t. Nqthm which supports neither higher-order logic nor mutual recursion. The use of a system such as HOL should probably be more recommended in that case.
- the second benchmark has shown that generalization can be less tedious with tail-recursive functions.

- with the first example, we have seen that there can be a great difference between the proof CPU times. This is not so clearly evident with the factorial example (times vary between 3 and 5 seconds).

Finally, the Nqthm version of the tail-recursive model is syntactically the simplest one, and translation to functions of that form is rather straightforward. It suffices to produce, for each state variable and output, its associated expression (by functional composition, starting from this element, up to primary inputs and state variables) and to put it at the right place in the function template. The translation principles are described in [Pi,91b] for the CASCADE language, they are quite similar for our synchronous VHDL subset.

For all these reasons, one can easily be convinced that this model is preferable as far as Nqthm is concerned. In fact, we have used this representation since the beginning of our work on synchronous systems, but it was an intuitive choice, without justification. In this paper, we have compared it with a more expressive one, we have proved that they are equivalent, and we have explained why our intuitive preference was the right one despite the better clarity of the pure recursive model. This study also demonstrates that Nqthm can be as useful as another proof tool for reasoning about synchronous systems. The choice of the theorem prover is not as important as the choice of the corresponding modelling strategy.

REFERENCES.

[BD,77] R.BURSTALL, J.DARLINGTON : "A transformation system for developing recursive programs". Journal of the ACM. Vol. 24, 1. January 1977.

[BF,93] J.M.BERGÉ, A.FONKOUA, S.MAGINOT, J.ROUILLARD : "VHDL'92 : The New Features of the VHDL Hardware Description Language". Kluwer Academic Pub., 1993.

[BK,91] F.P.BURNS, D.J.KINNIMENT, A.M.KOELMANS : "Correct interactive transformational synthesis of DSP hardware". Proc. European Design Automation Conference. Amsterdam (The Netherlands). 25-28 February 1991.

[BM,88] R.S.BOYER, J S.MOORE : "A Computational Logic Handbook". Perspectives in Computing, Vol. 23. Academic Press, Inc. 1988.

[BP,92] 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.

[Br,89] A.BRONSTEIN : "String-Functional Semantics and the Boyer-Moore Mechanization for the Formal Verification of Synchronous Circuits". PhD Thesis, Stanford University, Report n° STAN-CS-89-1293. December 1989.

[BT,89] A.BRONSTEIN, C.L.TALCOTT : "Formal Verification of Pipelines based on String-Functional Semantics". Proc. IFIP WG 10.2 Int. Workshop Nov. 1989. In "Formal VLSI Correctness Verification", L.Claesen Ed., North Holland, 1990.

[Ca,87] R.CAMPOSANO : "Structural Synthesis : Some References". Advanced Summer Course on "Logic Synthesis and Silicon Compilation for VLSI Design". L'Aquila (Italy). 13-18 July 1987.

[CP,88] P.CAMURATI, P.PRINETTO : "Formal Verification of Hardware Correctness: Introduction and Survey of Current Research". IEEE Computer, Vol.21, n°7. July 1988.

[CP,91] S.COUPET-GRIMAL, L.PIERRE : "Recursive Models for Synchronous Sequential Devices". Research Report IMAG/ARTEMIS n° 855-I, Grenoble (France). July 1991.

[Di,78] D.DIETMEYER : "Logic Design of Digital Systems". Allyn and Bacon. 1978.

[Go,80] M.GORDON : "The denotational semantics of sequential machines". Information Processing Letters, Vol.10, n°1. February 1980.

[Go,84] M.GORDON : "LCF-LSM". Technical Report n°41. Univ. of Cambridge (UK). 1984.

[Go,85] M.GORDON : "HOL : a machine-oriented formulation of higher order logic". Technical Report n°68. University of Cambridge (UK). May 1985.

[Gu,92] A.GUPTA : "Formal Hardware Verification Methods : a Survey". Formal Methods in System Design, Vol.1, 1992.

[GW,85] S.M.GERMAN, Y.WANG : "Formal Verification of Parameterized Hardware Designs". Proc. IEEE Int. Conf. on Computer Design : VLSI in Computers, October 1985.

[HB,89] W.A.HUNT, B.C.BROCK : "The Verification of a Bit-slice ALU". Workshop on Hardware Specification, Verification and Synthesis : Mathematical Aspects. Cornell University, Ithaca, NY (USA). 5-7 July 1989.

[HK,92] P.HARRISON, H.KHOSHNEVISAN : "A new approach to recursion removal". Theoretical Computer Science 93, 1992.

[HL,78] G.HUET, B.LANG : "Proving and Applying Program Transformations Expressed with Second-Order Patterns". Acta Informatica 11. 1978.

[Hu,86] W.A.HUNT : "FM8501 : A verified microprocessor". Institute for Computing Science, University of Texas, Austin (USA). Technical Report n°47. February 1986.

[Ie,88] IEEE Standard VHDL Language Reference Manual, IEEE. 1988.

[Jo,84] S.D.JOHNSON : "Synthesis of Digital Designs from Recursion Equations". The MIT Press, Cambridge. 1984.

[Jo,86] S.D.JOHNSON : "Digital Design in a Functional Calculus". In G.J.Milne and P.A.Subrahmanyam Eds., Proc. of the Workshop on Formal Aspects of VLSI Design. Elsevier Science Publishers, 1986.

[Ka,74] G.KAHN : "The Semantics of a Simple Language for Parallel Programming". Proc. IFIP Congress 74. Amsterdam (The Netherlands). 1974.

[KP,94] M.KAUFMANN, P.PECCHIARI : "Interaction with the Boyer-Moore Theorem Prover : a Tutorial Study using the Arithmetic-Geometric Mean Theorem", Technical Report n°100, Computational Logic Inc., August 1994.

[OD,87] J.T.O'DONNELL : "Hardware Description with Recursion Equations". Proc. Int. Conf. on CHDL'87, Amsterdam (The Netherlands). M.Barbacci and C.Koomen Eds., Elsevier Science Publishers, 1987.

[Pi,91a] 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 (USA). January 1991.

[Pi,91b] L.PIERRE : "From a HDL Description to Formal Proof Systems : Principles and Mechanization". Proc. 10th Int. Symposium CHDL'91. Marseille (France), April 1991. In "CHDL and their Applications", D.Borrione & R.Waxman ed., North Holland, 1991.

[Pi,94] L.PIERRE : "An automatic generalization method for the inductive proof of replicated and parallel architectures". Proc. International Conference on Theorem Provers in Circuit Design. Bad Herrenalb (Germany), September 1994.

[Ru,92] D.M.RUSSINOFF : "A Mechanical Proof of Quadratic Reciprocity". Journal of Automated Reasoning, Vol.8, n°1, 1992.

[Sh,85] N.SHANKAR : "A mechanical proof of the Church-Rosser theorem". Institute for Computing Science, University of Texas, Austin. Technical Report n°45. March 1985.

[VV,92] D.VERKEST, J.VANDENBERGH, L.CLAESEN, H.DE MAN : "A description methodology for parameterized modules in the Boyer-Moore logic". In "Theorem provers in circuit design", V.Stavridou, T.Melham & R.Boute ed., North Holland, 1992.

[Yu,90] Y.YU : "Computer Proofs in Group Theory". Journal of Automated Reasoning, Vol.6, n°3, 1990.

ACKNOWLEDGEMENTS.

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

I am grateful to the anonymous reviewers for their fruitful comments.