Theorem Provers in Circuit Design
V. Stavridou, T.F. Melham and R.T. Boute (Eds.)
Elsevier Science Publishers B.V. (North-Holland)

A Description Methodology for Parameterized Modules in the Boyer-Moore Logic †

D. Verkest[a], J. Vandenbergh[a], L. Claesen[a b], H. De Man[a b]

[a] IMEC, Kapeldreef 75, 3001 Leuven, Belgium

[a b] Professor at Catholic University Leuven, Kard. Mercierlaan, 3001 Leuven, Belgium

Abstract

We present a method to *hierarchically* describe *combinatorial* and *sequential* circuits in the Boyer-Moore logic. The description style resembles the style of traditional hardware description languages and should look quite familiar to hardware designers. In addition it allows an easy translation to existing hardware description languages.

Keyword Codes: B.7.1; F.4.3; I.2.3

Keywords: Integrated Circuits, Types and Design Styles; Formal Languages; Deduction and Theorem Proving.

1 Introduction

The description style we propose here was developed to formally describe, specify and verify a module library which is used by the CATHEDRAL silicon compiler suite [1, 2]. The module library comprises some 25 functional building blocks (FBBs) built up hierarchically from over 60 leaf cells. All the functional building blocks are parameterized in the word length. Some building blocks also have other parameters which indicate some particular structural composition of the module. The FBBs are the basic building blocks of the silicon compilers and hence the correctness of the synthesis result depends heavily on the correctness of the FBBs. This is the main reason we perform a formal and mechanical verification of all the FBBs in the library [5, 6] by means of the Boyer-Moore theorem prover [3].

At the moment the synthesis system provides two routes to layout. One makes use of standard cell layout techniques and in that case the *structure* of the FBBs is described in HILARICS [4] - a language used internally in IMEC for the description of parameterized structure. From the HILARICS descriptions standard cell layout of the instantiated FBBs

†Research sponsored by ESPRIT BRA 3216 CHARME

can be obtained. The other route makes use of custom layout for the FBBs and in that case the *structure and layout* of the FBBs is described in a module generation environment (MGE) [7].

Since the formal descriptions of the FBBs in Boyer-Moore logic cannot be used directly by the synthesis system we have implemented some tools to provide a link with the actual silicon compilers. The formal descriptions in the Boyer-Moore logic [3] are used to automatically generate parameterized descriptions in the HILARICS language for all the FBBs. From these descriptions standard cell implementations can be obtained. The formal descriptions are also used as a reference for net list comparison of the extracted layouts generated from the MGE descriptions of the FBBs [6]. Finally the Boyer-Moore descriptions are used as a starting point for the formal verification of synthesis primitives in the CATHEDRAL 2nd synthesis system [8, 9].

The traditional method for the description of digital synchronous circuits in the Boyer-Moore logic was introduced by Hunt [10] for the description of the FM8501 and its successors and consists of modeling the behavior of the design directly in the logic. Every hardware circuit is modeled by a function in the logic which has as formal parameters the input terminals of the circuit. A hierarchical design is built up by combining the various functions describing the components of the design. Circuits which are parameterized by the word length of their inputs are described using recursion. To model sequential designs the registers (or state variables) are included as additional parameters of the functions and are updated appropriately in every recursive call. Every recursion step of this "sequential" function corresponds to one clock tick.

Recently Hunt introduced a new method for describing circuits in the Boyer-Moore logic using list constants [11]. Several interpretation functions can be provided to give a semantic meaning to the list constants. The semantics of the HDL developed in this way are described using these interpretation functions. The obvious advantages of this approach are

- a description which can be made to resemble a conventional HDL more closely;
- the possibility of having several interpretation functions which describe different aspects of the design (behavior, timing, ...);
- the possibility to define transformations and prove them correct with respect to the interpretation functions.

The main drawback of this approach is an increase in complexity of the correctness proofs. In [11] it is mentioned that sequential circuits can not yet be modeled in the proposed HDL, but this is probably just a matter of time.

In this paper we will describe a method which allows to *hierarchically* describe the structure of *combinatorial and sequential* parameterized designs *directly in the logic*. The description template is developed to have the feel of a traditional structure description language such as the HILARICS language in use in IMEC. The method will make extensive use of the LET construct which is available as syntactic sugar in the Boyer-Moore logic. It will use association lists to assign names to inputs and registers of the cells being described. The function describing the circuit will return an association list with the value of the outputs and possibly new values for the registers.

The motivation for the development of this description style will be explained in the next section. In section 3 we will discuss the implications this description style has for the proofs of correctness with the theorem prover. Some conclusions are due in section 4.

2 Requirements for description and verification of a module library

Our first attempts [5] at modeling the module library followed Hunt's first approach closely. However, for the formal verification of a hierarchically constructed module library this approach has some disadvantages: it cannot deal with a variable number of terminals and it is impossible to describe sequential circuits hierarchically. Due to the mixture of behavior and structure in the definitions, the descriptions are difficult to translate to conventional HDLs such as HILARICS.

In the remaining of this section we will discuss these problems in more detail using a simplified version of the multiplexor FBB (see figure 1) and the serial-parallel register FBB (see figure 4) as an example. The simplified multiplexor `mux` has two parameters: the word length `nb` and a flag `nbus` to choose between a configuration with two or three input busses. A possible instance of the multiplexor is e.g. `(mux 24 T)` to create a multiplexor which can select one out of 3 input busses which are all 24 bits wide. The simplified version of the serial-parallel register `spreg` consists of a simple register and the multiplexor and it has three operation modes: initialization, hold and shift-up. This simplified version of `spreg` has only one parameter `nb` to indicate the word length of the operands.

2.1 HILARICS, a conventional HDL

The description style we propose in this paper is intended to look familiar to hardware designers and consequently to be easily translatable to a traditional HDL as HILARICS. In this first subsection we will therefore briefly discuss the major features of HILARICS and indicate some of the main differences of the language compared to the Boyer-Moore logic.

The language in use at IMEC for the structural description of parameterized modules is HILARICS. This language describes structure only i.e. hierarchical composition and interconnections of cells. A HILARICS description consists of four blocks:

1. A cell header definition containing the name of the cell, the parameters and their type information.

2. An interface definition declaring the inputs and outputs of the cell together with their type and possibly dimension.

3. A component definition block which declares the components being used and which indicates of which cell they are an instance.

4. A connection definition block which indicates how the different components are interconnected with each other and with the external terminals.

It is possible to use control constructs such as conditionals and for loops to describe the interface, component and connection block or parts thereof. HILARICS allows for net oriented as well as component oriented descriptions. Recursive descriptions are allowed but can only be used in the net oriented description style. An important aspect of HILARICS is that once a signal or terminal is declared to be of type bit vector, all references to that signal must explicitly include the dimensions of the signal.

The description style used in Boyer-Moore descriptions of the FBBs can probably be described best as *recursive* and *component oriented.* The main difference with a recursive and component oriented HILARICS description becomes obvious in the description of sequential circuits. Sequential circuits are modeled in Boyer-Moore logic by recursive functions. The registers or state variables are included as formal parameters of the function and they are updated in every recursion step. In that way every recursion step of the function corresponds to one clock tick of the global clock in the sequential circuit. The registers have to be included as formal parameters because the Boyer-Moore logic is purely functional and hence the only way to pass the next state value of the register to the next invocation of the recursive function is through the function's formal parameters. The register does not occur as a cell (function) in the body of the recursive function describing the sequential circuit. In HILARICS a register FBB is simply a block with input terminals and output terminals, just as any other piece of hardware. When translating the Boyer-Moore description of a sequential circuit to HILARICS the register block would have to materialize out of thin air.

To give Boyer-Moore descriptions of the FBBs a more familiar look to hardware designers and to make them easier to translate into HILARICS we'll have to meet the following requirements:

- Separation of structural parameters, physical inputs and state variables.
- Facilities to describe FBBs with a variable number of physical input terminals (such as the multiplexor) and state variables.
- Provide a way to describe a register as a function in the body of the recursive functions describing sequential circuits.
- Hierarchical modeling of sequential circuits.

The first two points will be discussed in the next section and the two last points in section 2.3.

2.2 Variable number of terminals

In the introduction we already mentioned that modules have parameters which control the exact composition of the module. As a consequence the number of inputs and outputs of a module may vary depending on the parameters. For example the multiplexor FBB can have two or three input terminals depending on the value of the `nbus` flag. Due to this parameterization it is not possible to describe these FBBs in the Boyer-Moore logic by a function with a fixed number of inputs.

The obvious solution to this problem is to use a list of inputs. The successive elements of the list represent the different input terminals of the cell. When a FBB has a parameterized number of inputs this will affect the length of the list **inputs** but it will not affect the number of formal parameters of the function describing the cell. To access a specific terminal of the cell one has to know the place of this terminal in the linear list **inputs**. The actual names of the terminals can be assigned to the appropriate nest of **CARs** and **CDRs** using the LET construct available as syntactic sugar in the Boyer-Moore system. Inside the body of the definition the terminals can then be referenced by their names as illustrated by the definition of the multiplexor in 2.1.

2.1

```
(defn mux (nb nbus inputs)
 (let ((cmux (CAR inputs))
       (ita (CADR inputs))
       (itb (CADDR inputs))
       (itc (CADDDR inputs)))
 (if (and (Numberp nb) (not (equal nb 0)) (Boolp nbus) (listp inputs))
     (if (equal nb 1)
         (if nbus
             (BV (a_mux3 (BVbit ita) (BVbit itb) (BVbit itc)
                         (BVbit cmux) (BVbit (BVvec cmux)))
                 (BVnil))
             (BV (a_mux2 (BVbit ita) (BVbit itb) (BVbit cmux)) (BVnil)))
         (if nbus
             (BV (a_mux3 (BVbit ita) (BVbit itb) (BVbit itc)
                         (BVbit cmux) (BVbit (BVvec cmux)))
                 (mux (sub1 nb) nbus
                      (list cmux (BVvec ita) (BVvec itb) (BVvec itc))))
             (BV (a_mux2 (BVbit ita) (BVbit itb) (BVbit cmux))
                         (mux (sub1 nb) nbus
                              (list cmux (BVvec ita) (BVvec itb))))))
     (BVnil))))
```

In this definition, BV is the basic constructor for the bit vector data type. It is axiomatized in the Boyer-Moore logic by an invocation of the shell principle as indicated in definition 2.2.

2.2

```
(add-shell BV BVnil BVp
 ((BVbit (one-of truep falsep) false)
  (BVvec (one-of BVp) BVnil)))
```

The result of (BV abool abv) is a new bit vector consisting of the original bit vector abv with the boolean abool appended in front of it. The invocation of the shell principle also axiomatizes the destructor functions BVbit which returns the first bit of a bit vector and BVvec which returns all but the first bit. Furthermore a recognizer function BVp is created. This function returns true on bit vector objects and false on all other objects. Finally the BVnil function represents the bottom object of the bit vector shell.

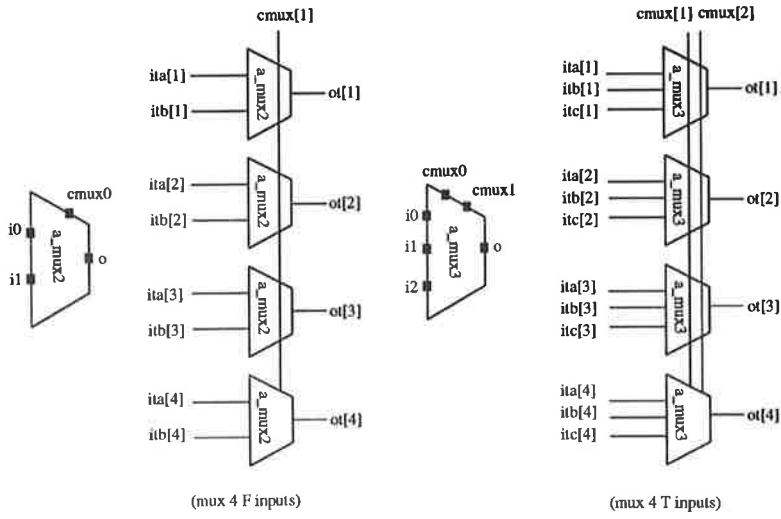

Figure 1: The simplified version of the multiplexor

The functions `a_mux2` and `a_mux3` model the basic multiplexor leaf cells with resp. 2 and 3 data inputs as can be seen in figure 1. The base case of definition 2.1 corresponds to input busses of size 1. In that case only one leaf cell is placed and depending on the value of `nbus` this will be `a_mux2` or `a_mux3`. In the recursive case the output is constructed with the `BV` function. The first bit is the output of the leaf cell applied to the first bits of the input busses. The remainder of the bit vector is constructed by a recursive call to `mux` with input busses which are now one bit smaller.

This kind of definition has a number of drawbacks:

1. To instantiate (call) the `mux` FBB one should be aware of the order in which the physical input terminals appear in the formal `inputs` list. In any conventional hardware description language it is sufficient to remember the names of the terminals.

2. It is not evident from the definition what the type is of the input terminals nor under which condition certain terminals are present in the FBB (e.g. when do we have an `itc` terminal).

3. It is not clear what the name or type is of the output terminals.

4. When an output of a certain component is used more than once, this component will occur twice in the description. This may pose problems when translating to HILARICS because one cannot decide whether this double occurrence was intentional or not. In addition components are not given names.

To solve problem 1 we will use an association lists for the inputs instead of a linear list. The order of the terminals in this association list is of no importance since the terminals

can now be referenced in the body of the definition by their names. Since hardware circuits can also have multiple output terminals, the functions describing the FBBs will also return association lists. We define this association list in Boyer-Moore logic in the following way (`assoc` is a builtin function of the Boyer-Moore logic which performs the same function as the Common LISP assoc function):

2.3

```
(defn net (term1 term2 nets)
 (cons (cons term1 term2) nets))

(defn term (name nets)
 (CDR (assoc name nets)))
```

The function `net` adds a pair with `term1` as key and `term2` as value to the association list `nets` which has the effect of making a connection from `term1` to `term2`. Note that in the Boyer-Moore descriptions there is always a sense of *direction* to the connections: `term1` is the name of an input terminal or an external output terminal and `term2` has to be the name of an output terminal or an external input terminal. The function `term` can be used to access the output terminal `name` from a component or to access an external input `name`.

Problem 2 will be solved by providing an explicit type checking function for the FBB which contains all the information concerning the parameters and input terminals of the FBB. For the multiplexor this will be:

2.4

```
(defn mux-inp (nb nbus inputs)
 (if (and (Numberp nb) (lessp 0 nb)      ; nb is a positive, non-zero integer
          (Boolp nbus)                   ; nbus is a Boolean
          (listp inputs))                ; inputs is an (association) list
     (and (BVtype (term 'ita inputs) nb)
          (BVtype (term 'itb inputs) nb)
          (if nbus
              (and (BVtype (term 'itc inputs) nb)
                   (BVtype (term 'cmux inputs) 2))
              (BVtype (term 'cmux inputs) 1)))
     F))
```

In this definition (`BVtype abv anum`) is a shorthand for establishing that `abv` is a bit vector of size anum. In Boyer-Moore terms:

2.5

```
(defn BVtype (abv anum)
 (and (BVp abv) (equal (BVsize abv) anum)))
```

In definition 2.4 the test of the conditional establishes the type of the structural parameters. If all of these parameters are of the correct type then the `mux-inp` function returns true when the conjunction in its then branch is true. This conjunction declares the type of the physical input terminals: terminals `ita` and `itb` are bit vectors of size `nb`. If `nbus` is true then `itc` is also a bit vector of size `nb` and the size of the `cmux` bit vector is 2. If

nbus is false there is no itc terminal and the size of the cmux bit vector is 1. If any of the parameter - terminals combination in the actual function does not satisfies these type restrictions then mux-inp returns false.

Problem 3 is solved by also using an association list for the outputs. In addition we will prove a lemma stating what the type of the outputs is. Obviously we can only prove this lemma after we have defined the function describing the actual mux FBB. For the case of the multiplexor this lemma looks like:

2.6

```
(prove-lemma mux-outp ()
 (implies (mux-inp nb nbus inputs)
          (let ((Cp (mux nb nbus inputs)))
               (BVtype (term 'ot Cp) nb))))
```

The lemma mux-outp states that the terminal ot of the mux FBB is a bit vector of size nb under condition that the parameters and input terminals of the FBB meet their type restrictions. This corresponds to the information concerning the output terminals which can be found in the interface block of the HILARICS description.

Finally problem 4 can easily be solved by requiring that any component declaration is assigned a name by means of the LET construct. If a certain component is intentionally duplicated it will occur in two different LET statements.

All this results in the following definition for the multiplexor:

2.7

```
(defn mux (nb nbus inputs)
 (if (mux-inp nb nbus inputs)
     (let ((Cp1 (if nbus
                    (a_mux3
                     (net 'i0 (BVbit (term 'ita inputs))
                     (net 'i1 (BVbit (term 'itb inputs))
                     (net 'i2 (BVbit (term 'itc inputs))
                     (net 'cmux1 (BVbit (term 'cmux inputs))
                     (net 'cmux0 (BVbit (BVvec (term 'cmux inputs))) ()))))))
                    (a_mux2
                     (net 'i0 (BVbit (term 'ita inputs))
                     (net 'i1 (BVbit (term 'itb inputs))
                     (net 'cmux0 (BVbit (term 'cmux inputs)) ()))))))))
      (if (equal nb 1)
          (net 'ot (BV (term 'o Cp1) (BVnil)) ())
          (let ((Cpr (mux (sub1 nb) nbus
                      (net 'ita (BVvec (term 'ita inputs))
                      (net 'itb (BVvec (term 'itb inputs))
                      (net 'cmux (term 'cmux inputs)
                      (if nbus
                          (net 'itc (BVvec (term 'itc inputs)) ())
                          ()))))))))
           (net 'ot (BV (term 'o Cp1) (term 'ot Cpr)) ()))))
    ()))
```

For the definition of more complex structures than this simple multiplexor, more powerful constructor and destructor functions than BV, BVbit and BVvec are required. These functions can be defined in terms of the basic BV shell and its destructor functions and we will come back to them in section 3 when we discuss the implications of the description style for the correctness proofs in the theorem prover. The mechanically generated HILARICS description corresponding to this Boyer-Moore definition can be found in the appendix.

2.3 Hierarchical description of sequential systems

When describing sequential circuits in the Boyer-Moore logic as recursive functions operating on streams of values it is impossible to describe circuits hierarchically. To illustrate this point we will make several alternative descriptions of some sequential circuits. For the sake of clarity these functions will operate on natural numbers rather than bit vectors and type checking will be reduced to a minimum. First we define a purely combinatorial function hadder and a multiplexor mux.

2.8

```
(defn hadder (input) (add1 input))

(defn mux (cmux inT inF) (if cmux inT inF))
```

The hadder function has only one input and always increments its input. The function mux has a control line cmux which determines whether the inT or inF input is returned at the output. We also define a conditional register regis as

2.9

```
(defn regis (input write reg)
 (if (listp input)
     (cons reg
           (regis (CDR input)
                  (CDR writo)
                  (if (CAR write) (CAR input) reg)))
     ()))
```

The function regis is a simple sequential block where input is a list representing the input value as it evolves over time, write is a list representing the value of the control line as it evolves over time and reg is the current value of the register. The function returns a list representing the output of the register as it varies over time. In each recursive call the register reg is updated: if the current value of the control line (CAR write) is true then the new value for reg is the current value of the external input, otherwise, we keep the current contents of the register.

We will now describe a simple sequential circuit, simple-seq, using the blocks we just defined. Its structure is shown in figure 2 and its definition is given below:

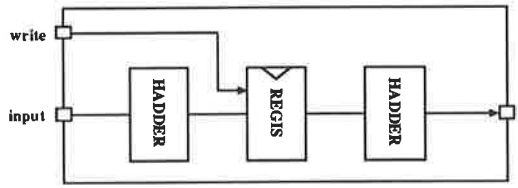

Figure 2: A simple sequential circuit

2.10

```
(defn simple-seq (input write reg)
 (if (listp input)
     (cons (hadder reg)
           (simple-seq (CDR input)
                       (CDR write)
                       (if (CAR write) (hadder (CAR input)) reg)))
     ()))
```

In a true hierarchical description style one would like to describe `simple-seq` without any knowledge of the internal structure of `regis`. This is not possible given the definitions used above. One has to be aware of the (conditional) next state function of `regis` to correctly describe `simple-seq`. In addition it is not obvious from the above definition what the exact structure of `simple-seq` is: the register does not occur as a component in the body of `simple-seq`.

A first correction in the description style consists of describing sequential systems by a function which takes a current value of the input, a current value of the register and returns a current value of the output and a next value for the register. Actually this is the way a sequential system is described in a *register transfer language*. This results in the following definitions for `regis` and `simple-seq`:

2.11

```
(defn regis (input write reg)
 (list reg (if write input reg)))

(defn simple-seq (input write reg)
 (cons (hadder (CAR (regis (hadder input) write reg)))
       (CADR (regis (hadder input) write reg))))
```

In contrast to the previous definition 2.9 of `regis` this definition gives a "static" view of the register. The formal parameters of the function are now natural numbers and not lists of natural numbers as in definition 2.9. The function returns a list consisting of the output value of the register (given by the current contents of the register `reg`) and a new value for the register which is either the external input or the current state depending on the control line `write`. In addition the internal structure of `simple-seq` is clear from the body of the definition: the external output of `simple-seq` is connected to the output of `hadder` which has as input the output of `regis` which in turn has the external input `write` and the output of the first `hadder` as inputs.

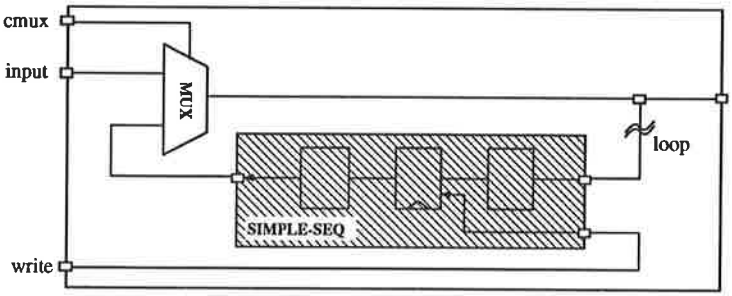

Figure 3: A sequential system with feedback

With this description style it is no longer necessary to open up the definition of regis to describe the `simple-seq` function. It is sufficient to know (1) that the register is in the block `regis` and (2) that the next state value is the CDR of the result of a sequential function. Compared with the description of `simple-seq` in 2.10 this new definition only describes the behavior of the circuit from one clock cycle to the next clock cycle. Consequently such a definition can only be used to reason about the correctness of the circuit from one clock cycle to the next (just as in register transfer languages). If at any time it is necessary to reason about the behavior of the circuit over more than one clock cycle a function has to be defined which maps the `simple-seq` function of 2.11 on a list of inputs while updating the registers appropriately. For a given sequential function `seq`, such a function *always* looks like

2.12

```
(defn seq-in-time (in-1 ... in-m reg-1 ... reg-n)
 (if (and (listp in-1) ... (listp in-m))
     (let ((RT (seq (CAR in-1) ... (CAR in-m) reg-1 ... reg-n)))
          (cons (CAR RT)
                (seq-in-time (CDR in-1) ... (CDR in-m)
                             (nth 1 (CDR RT)) ... (nth n (CDR RT)))))
     ()))
```

In the above template 2.12 the function `seq` is called once in every recursion step with as inputs the current value of the input terminals `(CAR in-i)` and the current contents of the registers. In the recursive call the registers are updated appropriately by selecting the correct next-state value from `(CDR RT)`.

The next example shows the description of a small system with feedback depicted in figure 3. To describe this circuit in a purely functional language as the Boyer-Moore logic one needs to cut the loop. Therefore we introduce an extra "state" in the circuit labeled `loop`.

2.13

```
(defn feedback (input cmux write loop reg)
 (let ((CS (simple-seq loop write reg)))
  (let ((CM (mux cmux input (CAR CS))))
       (list CM                                          ; current output value
             (cons CM                                    ; next-state for 'loop
                   (CADR (simple-seq CM write reg)))))))) ; next-state for 'reg
```

The function **feedback** returns a list of the current output value and of the new values for the state variables. The output of the circuit consists of the multiplexor **mux** which has one input connected to the **simple-seq** sequential block which in turn has as input the virtual state **loop**. The next state for **loop** is simply the output of the circuit and the next state for **reg**, the internal register of **simple-seq**, is obtained from the next-state of **simple-seq** but with as input the current output. The part of **feedback** dealing with the output allows to determine the structure of the cell except for the input of **simple-seq**. This is a special case because it takes as input **loop** - a state variable - which has no structural meaning. To find the actual connection for the input of **simple-seq** one has to look at the next-state function for **loop** and this turns out to be **mux** whose output is indeed connected to **simple-seq** in the real circuit.

Using the template 2.12 we can construct the function **simfeedback** to observe the behavior of this system in time.

2.14

```
(defn simfeedback (input cmux write loop reg)
 (if (and (listp input) (listp cmux) (listp write))
     (let ((RT (feedback (CAR input) (CAR cmux) (CAR write) loop reg)))
          (cons (CAR RT)
                (simfeedback (CDR input) (CDR cmux) (CDR write)
                             (CAADR RT) (CDADR RT))))
     ()))
```

Using the builtin evaluation environment (**r-loop**) of the Boyer-Moore system one observes the following behavior:

2.15

```
*(simfeedback (list 1 2 3 4 5 6 7 8 9)
              (list F F F F T T T T F)
              (list F T F T F T F T F)
              91 101)
 '(102 102 104 104 5 6 7 8 10)
```

We will now apply all the above to the sequential serial-parallel register FBB. We will also use an association list instead of a linear list to describe the registers. Therefore we define the functions **ns** and **reg** in analogy with **net** and **term**:

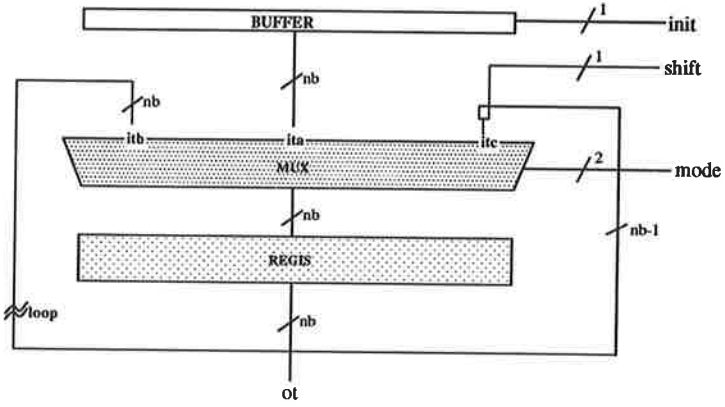

Figure 4: The structure of the serial-parallel register

2.16

```
(defn ns (reg1 reg2 states)
 (cons (cons reg1 reg2) states))

(defn reg (name states)
 (CDR (assoc name states)))
```

The theorem prover normalizes the body of a function before storing it in its data base. The description style we have proposed above makes frequent use of LET statements and due to the parameterization of the FBBs deeply nested if-then-else constructs also occur frequently. All this makes that the theorem prover often spends a lot of time in normalizing the body of the definition. We have experienced that splitting the output and next state functions of the FBB speeds up the acceptance of the function considerably and we therefore systematically use two functions to describe a sequential FBB: one has extension -O and describes the output function, the other one has extension -NS and describes the next state function.

The definition of the serial-parallel register of figure 4 then consists of three functions in the Boyer-Moore logic. The first one is the traditional type prescription function for parameters, inputs and internal registers. The second and third one describe the actual structure.

2.17

```
(defn spreg-inp (nb inputs states)
 (if (and (Numberp nb) (lessp 0 nb) (listp inputs) (listp states))
     (and (Boolp (term 'init inputs))
          (Boolp (term 'shift inputs))
          (BVtype (term 'mode inputs) 2)
          (BVtype (reg 'r states) nb)
          (BVtype (reg 'loop states) nb))
     F))
```

2.18

```
(defn spreg-0 (nb inputs states)
 (if (spreg-inp nb inputs states)
     (let ((buffer (buffer nb (net 'cin (term 'init inputs) ()))))
      (let ((mux (mux nb T
                      (net 'ita (term 'ot buffer)
                      (net 'itb (reg 'loop states)
                      (net 'itc (BVappend (BVvec (reg 'loop states))
                                          (BV (term 'shift inputs) (BVnil)))
                      (net 'cmux (term 'mode inputs) ()))))))))
       (let ((regis-0 (regis-0 nb
                               (net 'it (term 'ot mux) ())
                               (ns 'reg (reg 'r states) ()))))
        (net 'ot (term 'ot regis-0) ()))))
     ()))
```

2.19

```
(defn spreg-NS (nb inputs states)
 (if (spreg-inp nb inputs states)
     (let ((buffer (buffer nb (net 'cin (term 'init inputs) ()))))
      (let ((mux (mux nb T
                      (net 'ita (term 'ot buffer)
                      (net 'itb (reg 'loop states)
                      (net 'itc (BVappend (BVvec (reg 'loop states))
                                          (BV (term 'shift inputs) (BVnil)))
                      (net 'cmux (term 'mode inputs) ()))))))))
       (let ((regis-0 (regis-0 nb
                               (net 'it (term 'ot mux) ())
                               (ns 'reg (reg 'r states) ())))
             (regis-NS (regis-NS nb
                                 (net 'it (term 'ot mux) ())
                                 (ns 'reg (reg 'r states) ()))))
        (ns 'r (reg 'reg regis-NS)
        (ns 'loop (term 'ot regis-0) ()))))
     ()))
```

And again we can prove that the type of the output of spreg is a bit vector of size nb as is stated in the lemma spreg-outp.

2.20
```
(prove-lemma spreg-outp ()
 (implies (spreg-inp nb inputs states)
        (let ((Cp-0 (spreg-0 nb inputs states)))
             (BVtype (term 'ot (CAR Cp-0)) nb))))
```

The HILARICS description of this FBB is also included in the appendix.

3 Description and correctness proofs

In this section we will discuss the functions which are available for the description of the FBBs and the implications of the description style for the correctness proofs of the FBBs.

The definitions for `net`, `term`, `ns` and `reg` are non-recursive and the theorem prover will always open up these definitions. We do not want this because the correctness lemmas of the FBBs, which have these functions in their body, would be made useless. Therefore we disable these definitions. Of course this makes it impossible for the theorem prover to find out some basic facts as `(equal (term 'ita (net 'itb B (net 'ita A ()))) A)`. We therefore prove the lemma `term-of-net` which allows the theorem prover to reduce nests of `term` and `net` calls.

3.1
```
(prove-lemma term-of-net (rewrite)
 (equal (term name1 (net name2 term morenets))
        (if (equal name1 name2) term (term name1 morenets))))
```

This lemma will be applied repetitively on nests of `term` and `net` until a `net` call is found with `name1` as its first argument. A similar lemma can be stated and proved for the `ns` and `reg` definitions.

The basic data type used to describe the structure (and the behavior) of the FBBs is axiomatized by an invocation of the shell principle as shown in 2.2. This provides us with one constructor function `BV` and two destructor functions `BVbit` and `BVvec`. In order to easily describe more complex structures, a number of more powerful accessor functions has been defined of which the effect is illustrated in figure 5. Any application of a function from figure 5 to a bit vector can be translated into HILARICS. In some cases the translation tool needs to determine the size of the bit vector to which the function is applied in order to correctly describe the selection operation in HILARICS. For example `(BVfirstn 4 abv)` corresponds to `abv[size-4 .. 1]` in HILARICS when the word length of `abv` is `size`. The size of the bit vectors can usually be determined using the information found in the type restriction function for the FBB.

The functions of figure 5 are divided in two groups. The first one contains `BVbit`, `BVvec`, `BVfirstn` and `BVnvec` and we call it the MSB group because it starts with the selection of bits at the most significant bit (MSB) side of the bit vector. The second group contains `BVlast`, `BVbutlast`, `BVlastn` and `BVnbutlast` and is called the LSB group. Of course all these functions are ultimately defined (recursively) using the basic accessor and constructor functions introduced with the shell principle. Functions in the LSB group are mostly used to describe arithmetic functions or circuits such as adders. Functions of the

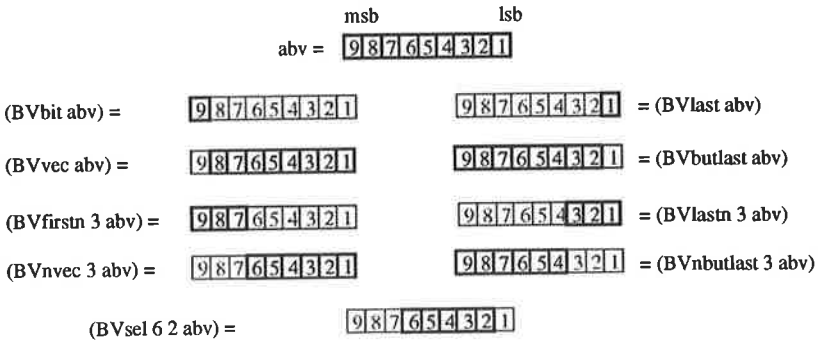

Figure 5: Access functions for the BV bit vector data type

MSB can be used to easily describe shifter structures. Any function in the LSB group could be replaced by one or more functions from the MSB group. The reason we keep these (redundant) functions around is that they often make it easier to describe particular structures.

The library of these functions and their properties is set up in such a way that the theorem prover can apply its elimination and generalization heuristics as well to functions from the MSB group as the LSB group. To achieve this all lemmas which are added automatically for the functions `BV`, `BVbit` and `BVvec` by the invocation of the shell principle are added and proved separately for `BVappend`, `BVlast` and `BVbutlast`.

When in a correctness proof functions of both the LSB and the MSB group are used we choose to rewrite all these functions into functions of one particular group. Normally when only functions of one group occur in proofs, we do not switch to functions in the other group because the theorem prover can apply its heuristics equally well to both groups.

A final note about the combination of the description style with the association lists and the above mentioned bit vector library. The theorem prover only carries out elimination of destructors on terms which are a first level function application. Therefore terms as `(BVvec abv)` where `abv` is a variable are amenable for destructor elimination. However terms as `(BVvec (term 'it inputs))` will never be subject to destructor elimination under the theorem prover's heuristics. The few cases where this would pose problems have to be solved by proving lemmas which deal explicitly with those cases.

4 Conclusion

We have presented a modeling method for describing digital synchronous circuits directly in the Boyer-Moore logic. This modeling method allows to *hierarchically* describe both combinatorial and sequential functions. Some syntactic sugaring functions were defined in the Boyer-Moore logic so that descriptions have the flavor of traditional hardware description languages. The translation of descriptions following this style into a traditional hardware description language is relatively easy. This description style is being used to

model the Cathedral II & III module library. A program which implements the translation from the Boyer-Moore descriptions of the modules to the equivalent HILARICS description is implemented.

In this modeling method, the `LET` statements and the `net` statements are still nested and thus there are only limited possibilities to share common parts of the description. Consequently the HILARICS code generated from these descriptions will be less compact than hand-written code. This is mainly due to the fact that the Boyer-Moore descriptions of the modules are component oriented whereas the HILARICS descriptions are net oriented. With the net oriented style it is much more easy to share the control structure of the description.

The functions defining the parameterized FBBs often have deeply nested if-then-else trees to deal with all the structural configurations in which the FBBs can occur. In combination with the sometimes complex type restriction function `FBB-inp` for the structural parameters and external inputs, this causes the generation of a large number of cases which have to be dealt with by the theorem prover. For non recursive definitions this combinatorial explosion can be kept in hand by some standard techniques described in chapter 13 of [3]. For recursive definitions we still need to find ways of keeping the explosion of cases under control.

References

[1] H.De Man, J.Rabaey, P.Six and L.Claesen, "Cathedral-II: A Silicon Compiler for Digital Signal Processing", *IEEE Design and Test*, December 1986, pp.13-25

[2] S. Note, W.Geurts, F.Catthoor and H.De man, "Cathedral-III: Architecture driven high-level synthesis for high throughput DSP applications", *Proc. 28th ACM/IEEE Design Automation Conference*, San Francisco CA, June 1991.

[3] R.Boyer, J.Moore, "A Computational Logic Handbook", *Academic Press Inc.*, 1988

[4] P.De Worm, R.Severyns, E.Willems, "HILARICS-2 User's Manual, v2.0",*Internal Report IMEC*, March 26, 1991

[5] D.Verkest, L.Claesen, H.De Man, "On the use of the Boyer-Moore theorem prover for correctness proofs of parameterized hardware modules", in *Formal VLSI Specification and Synthesis*, Ed. L.Claesen, Elsevier Science Publishers (North Holland), 1990, pp.99-116

[6] D.Verkest, L.Claesen, H.De Man, "Correctness proofs of parameterized hardware modules in the Cathedral-II synthesis environment", *Proc. of European Design Automation Conference*, Glasgow, Scotland March 1990, pp. 62-66

[7] P.Six, L.Claesen, J,Rabaey, H.De Man, "An intelligent Module Generation Environment", *Proceedings of the 23rd Design Automation Conference*, Las Vegas, June 1986, pp.730-735

[8] D.Lanneer et al., "Open-ended system for high-level synthesis of flexible signal processors", *Proc. European Design Automation Conference*, Glasgow, Scotland, April 1990, pp.272-276

[9] J.Vandenbergh, "Correct Library Development for High Level Application Oriented Synthesis in the Cathedral Environment", accepted for presentation at *IFIP International Workshop on Application Oriented Synthesis*, Dresden, March 23 - 25, 1992.

[10] W.Hunt, "FM8501: A Verified Microprocessor", Ph.D.Thesis, University of Texas at Austin, 1985

[11] B.Brock, W.Hunt, "The Formalization of a Simple Hardware Description Language", in *Formal VLSI Specification and Synthesis*, Ed. L.Claesen, Elsevier Science Publishers (North Holland), 1990, pp.83-98

Appendix: HILARICS description of multiplexor FBB

```
cell mux (nb, nbus)

int nb;
(0,1) nbus;

if (nb < 1) message("parameter nb has lower limit 1\n");

  input ita[nb..1];
  input itb[nb..1];
  if (nbus) {
     input itc[nb..1];
     input cmux[2..1];
  } else {
     input cmux[1];
  }

  output ot[nb..1];

end mux;

structure of mux

component {
  if (nbus) {
     cp1 : a_mux3 ();
  } else {
     cp1 : a_mux2 ();
  }
  if (nb != 1) {
     cpr : mux ((nb --), nbus);
  }
}

connection {
  if (nbus) {
     ita[nb] : net cp1.i0;
     itb[nb] : net cp1.i1;
     itc[nb] : net cp1.i2;
     cmux[2] : net cp1.cmux0;
     cmux[1] : net cp1.cmux1;
  } else {
     ita[nb] : net cp1.i0;
     itb[nb] : net cp1.i1;
     cmux[1] : net cp1.cmux0;
  }
  if (nb == 1) {
     ot[nb] : net cp1.o;
```

```
  } else {
     ita[(nb - 1)..1] : net cpr.ita[(nb - 1)..1];
     itb[(nb - 1)..1] : net cpr.itb[(nb - 1)..1];
     if (! nbus) {
        cmux[1] : net cpr.cmux[1];
     }
     if (nbus) {
        cmux[2..1] : net cpr.cmux[2..1];
     }
     if (nbus) {
        itc[(nb - 1)..1] : net cpr.itc[(nb - 1)..1];
     }
     ot[nb] : net cp1.o;
     ot[(nb - 1)..1] : net cpr.ot[(nb - 1)..1];
  }
}

end structure mux;
```

Appendix: HILARICS description of serial-parallel register FBB

```
cell spreg (nb)

int nb;

if (nb < 1) message("parameter nb has lower limit 1\n");

  input init;
  input shift;
  input mode[2..1];

  output ot[nb..1];

end spreg;

structure of spreg

component {
  buffer : buffer (nb);
  mux : mux (nb, 1);
  regis : regis (nb);
}

signal {
  loop[nb..1];
  mux_4[nb..1];
  buffer_3[nb..1];
}

connection {
  init : net buffer.cin;
  buffer_3[nb..1] : net mux.ita[nb..1], buffer.ot[nb..1];
  loop[nb..1] : net mux.itb[nb..1];
  loop[(nb - 1)..1] : not mux.itc[nb..2];
  shift : net mux.itc[1];
  mode[2..1] : net mux.cmux[2..1];
  mux_4[nb..1] : net regis.it[nb..1], mux.ot[nb..1];
  ot[nb..1] : net regis-o.ot[nb..1];
  loop[nb..1] : net regis-o.ot[nb..1];
}

end structure spreg;
```