A framework for VHDL combining theorem proving and symbolic simulation

Philippe Georgelin, Dominique Borrione, Pierre Ostier
Laboratoire TIMA, Grenoble, France
{Philippe.Georgelin, Dominique.Borrione, Pierre.Ostier}@imag.fr

Abstract

We present the status of an on-going work aiming at introducing symbolic simulation and theorem proving in a design flow that uses conventional description and simulation languages. This paper focuses on the formalization of the cycle simulation semantics of a synchronous subset of VHDL, in the ACL2 logic. The model is executable, and the results of its symbolic simulation can be proven equal to a specified expression. The ACL2 input is produced automatically from the VHDL source, which relieves the designer from an error prone manual translation.

1 Introduction

In the context of high-level synthesis, the industrial designers of signal processing circuits most often write the initial specification in an algorithmic language (C and its various extensions, Matlab, Mathematica, etc) and extensively simulate the behavior of this initial description. Then a (most often manual) synthesis step is performed, producing a Register Transfer Level description of the intended hardware processing module, in a conventional language (usually Verilog in England and the Americas, VHDL being more popular in continental Europe); this second description in turn is extensively simulated, and will eventually be the reference specification that all available automatic design tools will take as initial input. In fact, if modifications are made on the design, they are performed directly on the RTL description, and never traced back to the initial specification.

There is a large semantic gap between the initial behavioral specification, which computes some arithmetic function, and the RTL version of the circuit, which already introduces clock cycles, and views the computation in terms of a finite state machine. Popular automatic verification tools (model checkers, equivalence checkers) are applicable to the RTL description and below, but are inadequate to check that the RTL conforms to the initial specification.

The objective of our work is to develop methods and tools that can be introduced in the design flow to help validate the initial behavioral description, and check that the RTL description obtained after a high-level synthesis step conforms to it. Our approach consists in modeling the semantics of the standard design description languages in a formally manipulable model, and producing this model automatically, in order to relieve the designer from an error prone manual translation.

This paper presents the current status of an on-going project, whose long terms objective is to check the correctness of VHDL RTL descriptions synthesized from behavioral specifications. To this aim, we defined the simulation semantics of a synthesizable VHDL subset in the ACL2 logic, and built above the initial idea of J Moore [Moo98] to perform the value simulation, the symbolic simulation and the proof of functional equivalence on a single semantic model of the initial VHDL text.
An early approach was presented in [BGR00b]. In that paper, we macro-generated the ACL2 model from an intermediate format syntactically very close to VHDL, in fact in one to one correspondence at the statement level. That format was very readable, but the macro-generation did not scale well when we added more and more VHDL primitives, and in particular hierarchy. More precisely, the result of the macro-generation was directly fed into the ACL2 system, and it was difficult to control and correct it. In contrast, taking advantage of the I/O facilities of ACL2, it is possible to consider the intermediate format as a datum rather than a function, thus easier to manipulate, and already exhibit some essential semantic features of the description.

In the current approach, we no longer try to keep the VHDL syntax. From the VHDL description, an industrial compiler, LVS, provides an abstract syntax tree after lexical, syntactic and static semantic analysis. We start from this "verified" internal format (where *verified* is here understood in the VHDL acception of the word, meaning the set of static verifications that a compiler can perform), to extract the characteristics of the model, in the form of a set of property lists. The ACL2 model is then generated from this new intermediate format. The details of our formalization method for VHDL have been introduced in [VR00, BGR00a].

The earlier works could be considered a shallow embedding of VHDL in ACL2, since we relied on LVS to detect syntactic and static semantic errors (such as typing and scope of declared objects). Our current approach is even less an embedding of VHDL: it rather consists in providing the "elaboration" (again a VHDL word) of a VHDL description in view of its interpretation with ACL2. This attitude is motivated by the complexity of VHDL, and should be contrasted to the deep embedding of a simpler language such as

DUAL-EVAL [Hun00] (where concurrent sequential processes are absent, to the knowledge of the authors).

We implemented an interface on the ACL2 theorem prover, which offers commands to perform:

1. the automatic translation between VHDL and the logic of ACL2.
2. the simulation of numeric values
3. the symbolic simulation, which uses the simplification engine of ACL2
4. the proof that a given model variable, after symbolic simulation, holds a specified content.

This paper is structured as follows. Our model and the translator are described in the next section. The third section discusses simulation and symbolic simulation. The last section shows how to prove some properties.

2 Our formalization method

We model the *behaviour* of a VHDL description in the ACL2 input language. We restrict ourselves to a standardized synthesis subset of VHDL [IEE00]. We add the simplifying restriction that all processes are synchronized on the same clock edge; we can thus identify simulation cycle and clock cycle, and need not represent the clock event explicitly.

2.1 The VHDL subset

This section adopts the VHDL terminology, and assumes the reader to be familiar with the essential concepts of the language.

According to the standard, the synthesis subset excludes physical time and non-discrete types. We further limit the subset to single clock synchronization, without asynchronous statements (such as set and reset). We recognize the primitive types *bit*, *boolean*, *integer* (scalars and vectors) as well as enumerated user defined types.

A circuit is described by an *entity*, which declares its interface signals (recognized directions are *in* and *out*). At most one of these signals is the master clock. At least one *architecture* is associated with the *entity*, and describes the behavior and/or the structure of the circuit. Inside the *architecture*, concurrent processes may communicate through locally declared signals; to guarantee determinism, "shared" variables are excluded from the synthesizable subset, (i.e. variables may only be declared local to a process).

User defined subtypes and *pure* functions are recognized, however we do not accept *resolved* types for the moment. One or more instances of component may be declared and interconnected in the architecture.

According to the VHDL definition, all the *concurrent statements* in an *architecture* (signal assignments, assertions, procedure calls, blocks) are translated to an equivalent process; we thus only consider processes here. We further assume that all processes have been put in a normal form, with a single *wait* statement written:
wait until clock-edge.

Inside a process, except for declarative statements, we shall only discuss sequential *signal* assignments, *variable* assignments and *if* conditionals.

In this paper, we shall use the following toy description as running example. The entity **mysystem** starts computing the factorial of input **arg** when **start** is equal to '1'. The result is obtained after a varying number of cycles of the master clock **clk** : the computed value is then assigned to output **result** and output **done** is set to '1'.

The behavioral architecture **fact** is described as two concurrent sequential processes: **mult** which models a multiplier, and **doit** which is the control automaton.

```
entity mysystem is
    port (input : in natural;
  start,clk : in bit;
  output : out natural;
  done : out bit);
end mysystem;

-- purpose: factorial of n with 2 processes
architecture fact of mysystem is
    signal op1,op2,resmult : natural;
    signal startmult,endmult : bit;
begin

    Multiplier : process    -- process Multiplier
    begin
        wait until clk='1';
        if startmult='1' then
          resmult <= op1*op2;
        end if;
        endmult <= startmult;
    end process Multiplier;

    Doit : process     --control process
        variable mystate : natural := 0;
        variable R,F : natural := 0;
    begin
        wait until clk='1';
        if mystate = 0 then
          R := input;  F := 1;
          if start='1' then mystate := 1 ; end if;
        else
          if mystate = 1 then
```

```
            if R = 1 then
               output <= F;  done <= '1';  mystate := 0;
             else
                startmult <= '1';
                op1 <= R;  op2 <= F;   mystate := 2;
             end if;
          else
             if mystate = 2 then
                if endmult = '1' then
                   F := resmult;  R := R-1;
                   startmult <= '0';  mystate := 1;
                end if;
             end if ;
          end if;
        end if;
    end process Doit;
end architecture fact;
```

2.2 The memory state

An entity-architecture pair is formalized as an abstract state machine. A state represents a snapshot of the system interface pins and memory elements, and the architecture function maps a state to the next state. We call it the *memory state*.

The state is a list of values for all the signals and variables declared in the description. In VHDL, a **signal** has one current value, and one "driver" per process that assigns it. With the restrictions made above, it suffices to implement a single "next" signal value for local and output signals. Therefore, a signal is represented by two elements in the memory state. Input signals cannot be modified in the architecture, only their current value is present in the memory state. If **Sig** is a declared signal, in the machine state **Sig** refers to its current value and **Sig+** to its next value. A VHDL **variable** has no driver, and assignments modify its current value; it is represented by a single element in the memory state.

Unique naming is ensured by prefixing the identifiers of the declared objects with the identifier of their enclosing block (process or component).

Example of a memory state:

```
(arg start clk result done ;current interface signals
 op1 op2 resmult startmult endmult ;current local signals
 result+ done+ ;next output interface signals
 op1+ op2+ resmult+ startmult+ endmult+ ;next local signals
 doit.mystate doit.r doit.f) ;variables declared in doit
```

To access and modify the state elements of the model by their name rather than by their position in the list, the following functions are generated. All the functions are relative to a given entity-architecture pair; unique naming is ensured by prefixing the function name with the concatenation of the entity and architecture identifiers.

(defun **mysystemfact-get-nth** (var) *;returns the position of indicated variable*

```
    Type declaration
    (cond ((equal var 'arg) 0)
          ((equal var 'start) 1)
    ...)))

(defun mysystemfact-getst (var st) ;gets the value of a state element by name
  Type declaration
  (nth (mysystemfact-get-nth var) st))
(defun mysystemfact-putst (new var st) ;modifies the value of a state element
  Type declaration
  (update-nth (mysystemfact-get-nth var) new st))
```

We use type declarations and guards optimizations to increase simulation performances [WGH98]:

```
(declare (type (member arg
                start op1 op2 resmult startmult endmult
                op1+ op2+ resmult+ startmult+ endmult+
                doit.mystate doit.r doit.f res done res+ done+)
                var)
                (xargs :guard t))
```

After the generation of the accessors to the individual elements of the memory state, the recognizer predicate of the memory state is written. It contains the type of all the memorizing elements : bit, integer, array, signed, etc.
We guarantee that the memory state is well-founded after modifications.

Function `architecture-entity-name-MAKE-STATE` constructs the initial memory state, with all default values to variables and signals according to their declaration.

Function `arch-entity-name-UPDATE-ST` updates the memory state at the end of each simulation cycle, by copying the new value sig+ of each internal and output signal to its current value sig.

```
(defun MYSYSTEMFACT-UPDATE-SIGNALS (st)
(seq st
     (mysystemfact-putst 'op1 (mysystemfact-getst 'op1+ st) st)
     (mysystemfact-putst 'op2 (mysystemfact-getst 'op2+ st) st)
     (mysystemfact-putst 'resmult (mysystemfact-getst 'resmult+ st) st)
          ...
     (mysystemfact-putst 'done (mysystemfact-getst 'done+ st) st)))
```

2.3 Functions that model the VHDL simulation cycle

Functions are generated for all the concurrent statements of the VHDL description: concurrent signal assignments and processes. All these functions take a state as argument and produce a state as a result, actually construct a new state. For example, to model an assignment to a state element, we generate a new state where the new value replaces the old value.

Within the function generated for a process, statements are sequential as in the VHDL semantics. To this aim, as well as for efficiency reasons, we use the macro "seq" which computes sequentially a "single-threaded" memory state.

The translation of the MULT process is shown below.

```
;; This function represents the process MULT ;;;;;;;

(defun MYSYSTEMFACT-MULT-CYCLE (st)
(seq st
     (if (= (mysystemfact-getst 'startmult st) 1)
       (seq st
            (mysystemfact-putst 'resmult+
                                (* (mysystemfact-getst 'op1 st)
                                   (mysystemfact-getst 'op2 st)) st))
        st)
     (mysystemfact-putst 'endmult+
                         (mysystemfact-getst 'startmult st) st)))
```

One simulation cycle is performed by the `entity-arch-CYCLE` function, which calls all the process functions. The nesting order has no influence on the resulting state, as no two processes assign the same state element.

Function `entity-arch-SIMUL` performs N simulation cycles, by repetitively invoking the CYCLE function, and updating the current value of all output and local signals with their new value at the end of the cycle.

```
(defun MYSYSTEMFACT-CYCLE (st)
 (seq st (mysystemfact-mult-cycle st)
         (mysystemfact-doit-cycle st)))

(defun MYSYSTEMFACT-SIMUL (n st)
   (if (zp n)
      st
      (mysystemfact-simul (1- n)
                          (mysystemfact-cycle
                              (mysystemfact-update-signals st))))
)
```

2.4 ACL2 functions for components

Like processes, components are modeled by a transition function. This function passes the actual port values to and from the component state, and performs one step of the component simulation cycle by calling the CYCLE transition function of the entity-architecture pair bound to the component.

The memory state of the global architecture contains all memory states of components. In Lisp, component memory states are lists inside the global list.

Assume the architecture `fact2` for `mysystem` is written in a more structural style, where `mult` is an internal component rather than a process.

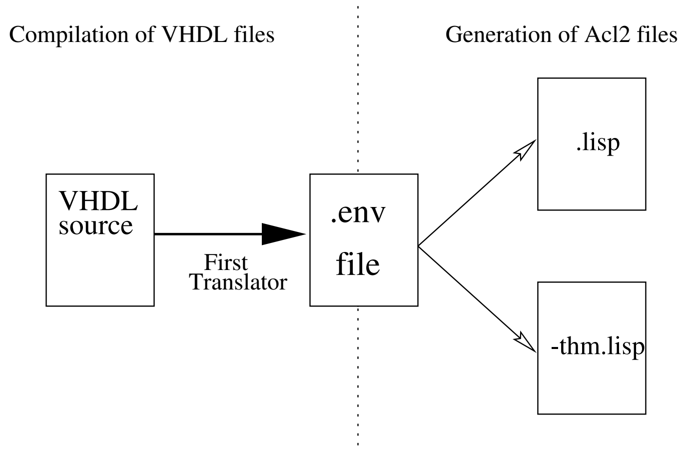

Figure 1: VHDL to ACL2 translators

```
--structural architecture with 2 components
architecture fact2 of mysystem is
    signal op1,op2,resmult : natural;
    signal startmult,endmult : bit;

    component Multiplier
       port ( clk, startmult: in bit; op1,op2 : in natural;
             endmult: out bit; resmult : out natural);
    end component;

begin
    Multi: Multiplier
       port map ( clk, startmult, op1,op2, endmult, resmult);

    Doit : process --control process as before
     ...
    end process Doit;
end architecture fact2;
```

In this case, the model state would be composed of:

```
(arg start
op1 op2 resmult startmult endmult
op1+ op2+ resmult+ startmult+ endmult+
(clk startmult op1 op2 endmult resmult endmult+ resmult+)
doit.mystate doit.r doit.f)
```

Models of architectures with components can be used for execution, symbolic simulation and formal verification exactly as detailed in the next sections. Our memory state is compositional.

2.5 The translator between VHDL and ACL2

The translator is divided into two phases :

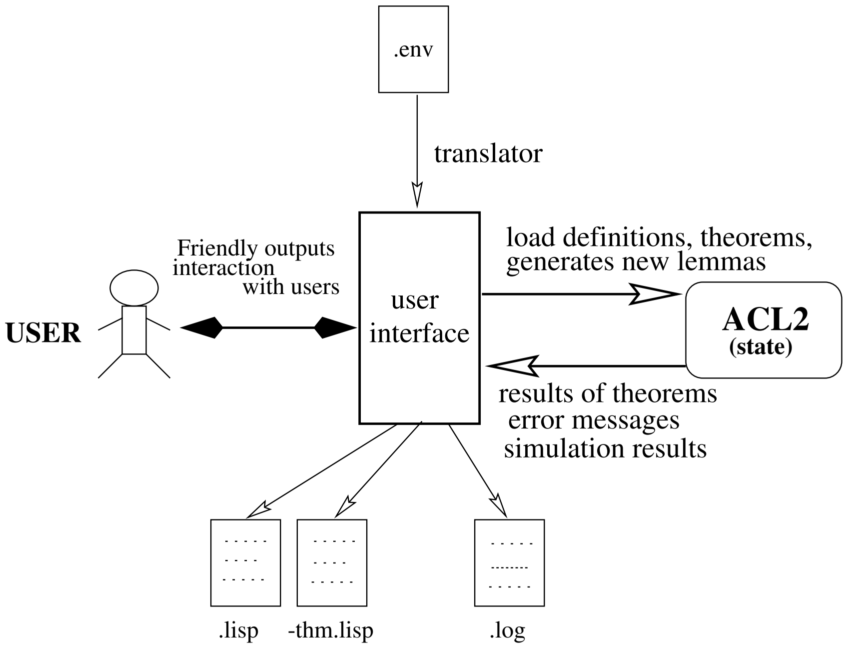

Figure 2: The role of the user interface

- a semantic extraction, which transforms the VHDL description (.vhdl) into a readable intermediate format, in the form of a list of its characteristic features (file entity-arch.env). The set of possible features is predefined, and each one is introduced by a key word. This format would fit with little modification other cycle-level simulation languages, such as e.g. a synthesizable subset of Verilog. (see Figure 1).
- A model generation, which takes the environment file and generates 2 files:
  - A .lisp file which contains all the ACL2 formalization in terms of functions.
  - A -thm.lisp which contains some of intermediate lemmas used for simplifying or for decomposing terms.

2.6 The user interface

A *user interface* (Figure 2) interacts with the top-level ACL2 read-eval-print loop and the user. It contains the second translator to produce .lisp and -thm.lisp. During executions, the tool produces a (or few) .log files, theses files are for debugging, they inform the user about functions calls and simplifications (log-output) or about proofs (log-proof). An example of interaction with the user interface is presented in the next section.

3 Numeric and symbolic simulation

3.1 Implementation of a model simulator

The generated model can be tested by entering numeric or symbolic values as input.

An example of numeric simulation is performed below. Dots abbreviate long lists of printed state values. The user inputs are captured after symbol greater than '>'.

```
----------------------------------------------
              ===================
            ((  SIMULATION MENU  ))
              ===================

   1 - Run numeric or symbolic simulation
   2 - Print memory state
   3 - Edit memory state
   4 - Reset memory state


q:return to MAIN MENU
----------------------------------------------
loaded : MYSYSTEM_FACT
Vhdl-ACL2>3
--------------- MYSYSTEM FACT-----------------
input signals: ARG and START
generic parameter:
local signals: OP1, OP2, RESMULT, STARTMULT and ENDMULT
variables: MYSTATE, R and F
----------------------------------------
Enter values like :(var1 val1 var2 val2....)
Modify>(arg 12 start 1)
  ARG            : 12
  START          : 1
  OP1            : 0
  ...
  DONE+          : 0

----------------------------------------------
              ===================
            ((  SIMULATION MENU  ))
              ===================

   1 - Run numeric or symbolic simulation
   2 - Print memory state
   3 - Edit memory state
   4 - Reset memory state


q:return to MAIN MENU
----------------------------------------------
loaded : "MYSYSTEM FACT"
Vhdl-ACL2>1

How many simulation cycle : 15

  ARG            : 12
  START          : 1
  OP1            : 8
```

```
OP2           : 11880
RESMULT       : 11880
STARTMULT     : 1
ENDMULT       : 0
OP1+          : 8
OP2+          : 11880
RESMULT+      : 95040
STARTMULT+    : 1
ENDMULT+      : 1
MYSTATE       : 2
R             : 8
F             : 11880
RES           : 0
DONE          : 0
RES+          : 0
DONE+         : 0
```

3.2 Symbolic simulation

For a first verification, the user can performs *symbolic simulation*. Symbolic simulation is the execution of an ACL2 model with actual inputs replaced by *symbolic* values. These values are mathematical variables (x, y ...), possibly restricted by conditions, representing arbitrary values. This method produces symbolic expressions in the designs outputs which describe their functional relation with the initial input values. Thus a single symbolic simulation run may stand for a very large or infinite number of test cases.

The routines provided by the expander generate theorems and simplifying expressions, under given assumptions.

We use the "expander.lisp" book written by Matt Kaufmann. The event `symsim` simplifies given terms and hypothesis.

The syntax of symsim is the following :

(**symsim** *call_of_function_to_be_simplified*
(*hypothesis*
))

For example, to perform symbolic simulation of the factorial model:

```
ACL2 !>(symsim (mysystemfact-simul 12 st)
               ((equal st (mysystemfact-make-state :arg q :start 1))
                (integerp q) (>= q 0)))
```

Some arithmetic rewrites rules are used for rewriting algebraic expressions (commutativity, associativity...). The user can disable them. So he can control the form of outputs of symbolic simulation.

We produce two outputs files for debugging purposes when the symbolic simulation fails:

- output.log : This file is the redirction of the "standard-Channel-Output", the contents of which is, by default, send to the user in the ACL2 prompt. We orient this channel into a file to debug more easily.

- proof.log : Same principle as above but associated to proof outputs.

More precisely, when we want perform symbolic simulation, we state numeric or symbolic values, and start from an initial state, say *st0*. Assume we have an entity-architecture pair called "entarch" and we perform a symbolic simulation for 3 simulation cycle. The simplification heuristic expands the call :

```
(entarch-simul 3 st0)
```

(using a rewrite rule) into :

```
(entarch-simul 2 (entarch-cycle (entarch-update-signals st0)))
```

And, so on :

```
(entarch-cycle (entarch-update-signals
                  (entarch-cycle (entarch-update-signals
                      (entarch-cycle (entarch-update-signals st0))))))
```

The simplification expands the definitions of `entarch-cycle` and `entarch-update-signals`.

After the simplification, we have a succession of *nth* and *update-nth*.

At this level, the simplification heuristics of ACL2 2.6 associated with the nu-rewrite algorithm, give a lot of *characteristic properties* of **nth** and **update-nth**.

The *characteristic properties* of `nth` and `update-nth` are listed below, where `i`, and `j` are element indexes. Properties P1 and P2 describe the access to an updated state. Property P3 indicates that only the last update to a variable matters. Property P4 swaps updates to distinct variables.

P1: `(nth i (update-nth i a st))` = `a`
P2: `i` $\neq$ `j` $\rightarrow$ `(nth i (update-nth j a st))` = `(nth i st)`
P3: `(update-nth i a (update-nth i b st))` = `(update-nth i a st))`
P4: `j` $<$ `i` $\rightarrow$ `(update-nth i a (update-nth j b st))` =
`(update-nth j b (update-nth i a st))`

From the properties above, ACL2 generates rewrite rules that reduce a nested expression `(update-nth` i_0 a_0 `(put` i_1 a_1 `(...` st`)))` to a *unique normal form* where there is at most one update for each variable, and updates are ordered by variable indexes. These properties also generate rules to read the value of a variable from such expressions. Thus this form is taken as the representation of states for proofs and symbolic simulation.

So, results of symbolic simulation are typically in the form :

```
(update-nth 2
 (binary-+ -1 q)
 (update-nth
  3
  q
  (update-nth
   4
   (binary-+ (unary-- q) (binary-* q q))
   (update-nth
    5
    1
    (update-nth
     6
     1
     (update-nth
      7
      (binary-+ -1 q)
      ...
         (update-nth 16 0
          st)))))))))))))))
```

This form is sent by the prover to the user interface, which transforms it and gives a more human-readable representation. We chose to display only the modified signals and variables in the symbolic simulation results.

```
  OP1          : (+ -3 q)
  OP2          : (+ (* 2 q) (- (* q q)) (- (* 2 q q)) (* q q q))
  RESMULT      :
(+ (- (* 6 q))
   (* 2 q q)
   (* 3 q q)
   (* 6 q q)
   (- (* q q q))
   (- (* 2 q q q))
   (- (* 3 q q q))
   (* q q q q))
  ...
```

4 Proofs of properties

Symbolic simulation expressions can be very unreadable, user can disable some runes (e.g commutativity-of-*, etc ...), but a better alternative is to give the expected result as a theorem. The interface creates the theorem and submits it to the prover.

The scheme is the following :

```
(thm
     (implies (and (equal (nth 0 st) q)
                   (equal (nth 1 st) 1)
                   (integerp q) (> q 13))
     (equal (mysystemfact-getst 'f Symbolic_result)
     itshape Expected_result))
```

```
    :hints (("Goal" :in-theory (disable nth update-nth))))
```

```
---------------------------------------------
              ==================
            ((  SYMBOLIC MENU  ))
              ==================

   1 - Print constraints
   2 - Print Runes
   3 - Remove Runes
             ---
   4 - Prove something

q:return to SIMULATION MENU
---------------------------------------------
loaded : "MYSYSTEM FACT"
Vhdl_ACL2>
4

Enter memory element to prove (or return): DOIT.F
Enter expression: (* q (- q 1) (- q 2) (- q 3))

...

Proof succeeded.

Congratulations ;-) Your property is True.
```

5 Conclusion

Our aim was to give the reader a flavor of our approach to apply theorem proving techniques in a design flow based on conventional hardware description languages.

Our formalization method consists in defining the simulation semantics of the appropriate design language in the ACL2 logic. We applied it to a synthesizable VHDL subset, the principles would be similar with another language such as Verilog. This formalization allows VHDL descriptions to be numerically and symbolically simulated by ACL2. Theorem proving techniques are useful in simplifying symbolic expressions, and showing that the content of designated state elements or outputs after a number of computation cycles is equal to the expected expression. We have implemented a compiler for a subset of VHDL to the ACL2 logic to formally analyze more realistic examples, such as submodules in a circuit which performs an IVT Reconstruction Operator [HSB+99] All those techniques are embedded inside a unique framework.

Up to now, only a small subset of VHDL is fully formalized and handled by the prototype. But it allows us to do some experiments and serves to demonstrate a possible use of the theorem prover.

Future extensions of this work include the proof of generic VHDL descriptions, and the proof of correctness of designs described using distinct specification and implementation description languages.

References

[BGR00a] D. Borrione, P. Georgelin, and V. Rodrigues. Symbolic simulation and verification of VHDL with ACL2. In *International Conference on HDL (HDLCONF'2000)*, pages 167–182, San Jose, 2000.

[BGR00b] D. Borrione, P. Georgelin, and V. Rodrigues. Using macros to mimic VHDL. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer Aided Reasoning: ACL2 Case Studies*, pages 167–183. Kluwer Academic Press, 2000.

[HSB+99] M. Harrand, J. Sanchez, A. Bellon, J. Bulone, A. Tournier, O. Deygas, J.C Herluison, D. Doise, and E. Berrebi. A single-chip cif 30-hz, h261, h263, and h263+ video encoder/decoder with embedded display controller. In *IEEE Journal of solid-state circuits*, Nov. 1999.

[Hun00] W. Hunt. The DE Language. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer Aided Reasoning: ACL2 Case Studies*, pages 151–166. Kluwer Academic Press, 2000.

[IEE00] IEEE Computer Society W.G. 1076.6. *IEEE Standard for VHDL Register Transfer Level Synthesis*, March 2000. http://www.eda.org/siwg.

[Moo98] J S. Moore. Symbolic simulation: An ACL2 approach. In *FMCAD'98*, pages 334–350, 1998. LNCS 1522.

[VR00] P. Georgelin V. Rodrigues, D. Borrione. An acl2 model of vhdl for symbolic simulation and formal verification. *XIII Symposium on Integrated Circuits and Systems Design (SBCCI'00),Manaus, Amazonas, Brazil*, September 18-22, 2000.

[WGH98] M. M. Wilding, D. A. Greve, and D. S. Hardin. Efficient simulation of formal processor models. Technical report, Advanced Technology Center, Rockwell Collins Avionics and Communications, Cedar Rapids, IA 52498, 1998. http://pobox.com/users/-hokie/docs/efm.ps.