Using the ACL2 Theorem Prover to Reason about VHDL Components

Vanderlei Moraes Rodrigues*

Dominique Borrione**

Philippe Georgelin**

Abstract

ACL2 is a theorem prover which uses an applicative subset of Common Lisp as specification language, and employs a quantifier-free first order logic to reason about these specifications. We define how to build an ACL2 model of a design described in a synthesizable VHDL. Using this single model, we may execute the design (which corresponds to standard simulation), perform a symbolic simulation of this design, and formally verify its properties. To handle designs employing components, we use abstract functions to represent an unspecified surrounding environment. This environment stands for the (unknown system) where the component is inserted. The ACL2 construction `encapsulate` is used to introduce such abstract functions. This technique allows for compositional reasoning, since component properties became available to the surrounding environment without the need to repeat the proofs for each component instance.

Keywords: formal verification of digital systems, VHDL, automated theorem proving, ACL2.

*Instituto de Informática, Universidade Federal do Rio Grande do Sul
e-mail: vandi@inf.ufrgs.br
**Laboratoire TIMA, Université Joseph-Fourier
e-mail: {dominique.borrione,philippe.georgelin}@imag.fr

1. Introduction

ACL2 [KMM00, KM97] is the automated theorem prover that follows the highly successful NQTHM [BM88] prover of Boyer and Moore. It uses an applicative subset of Common Lisp as specification language. Thus a system and its properties are modeled as a standard program in an ordinary programming language, rendering this model *executable* and, due to the existing compilers for Lisp, its execution is *efficient*. As a consequence, many system developers find modeling in ACL2 quite easy.

To reason about these models, ACL2 uses a quantifier-free, first-order logic. Its inference engine orchestrates a large number of proof techniques, including *induction*, in a highly automated and efficient way. Therefore, ACL2 is able to perform reasonably large proof steps by itself, relieving the user from providing more detailed proof scripts, in contrast to other theorem proving tools such as HOL [GM93] or PVS [Sha96].

We use ACL2 in the verification of *high-level, behavioral* descriptions of hardware designs written in VHDL [IEE94]. This is a large and complex language, whose official semantics is an informal operational description of the design simulator. The most popular verification techniques (such as finite model-checking [CGP00, McM93]) are not very appropriate for such descriptions because they present large or infinite state spaces, parametric and regular structures, separate components, etc. Such features are better described with more powerful formalisms and tools such as ACL2.

We develop a model for VHDL designs in ACL2, and build the translator that generates it automatically. This model is a set of functions describing the design simulation, and a supporting set of theorems. It is a *single* model which may be used for three distinct purposes. Since it is composed of Lisp functions, it can be *executed*, reproducing the behavior of a standard VHDL simulator.

The ACL2 model can also be used for *symbolic simulation*, which corresponds to executing the design with symbolic values as inputs, producing symbolic expressions as outputs. The inputs are mathematical variables (x, y, ...), possibly restricted by conditions, representing arbitrary values, and the outputs describe their functional relation with the inputs.

As in formal verification, a symbolic simulation run may correspond to a large or infinite number of cases. As in standard simulation, this method is fully automated. Therefore, symbolic simulation is a practical bridge between standard simulation and formal verification. The ACL2 model includes design specific theorems that transform the prover engine into a high-quality symbolic simulator.

On top of these two tasks, the ACL2 model is aimed at *formal verification*. In this case, properties of the VHDL design are stated and proved as theorems of the ACL2 model using the prover engine. To aid this task, the model also includes theorems stating design specific properties and supporting particular verification methods. We extend this set of theorems as we develop new verification methodologies or handle a

new class of designs.

Our ACL2 model of a VHDL design is *compositional*: the properties of a composite system may be derived from its component properties, allowing for modular system development and verification. Component properties and specifications may include arbitrary restrictions on the environment, which are checked when the component is used, and we may verify that components are functionally equivalent under environment assumptions. Equivalent components are interchangeable, and a composite system may know only the properties of its components.

Compositional reasoning about VHDL designs is not obvious due to the semantics of VHDL. To simulate a design, all its components are expanded into basic processes, and each process contributes to the simulation cycle in several distinct moments (process activation, signal updating, etc). In our approach, we embed a component into a partially specified *abstract system*, and ensure that properties hold in this arbitrary environment, allowing for the verification of reactive systems. This approach is based on the method adopted in UNITY [CM88] and other temporal logics.

This paper is organized as follows. Section 2 introduces the ACL2 model of a flat VHDL design. Section 3 studies VHDL components. The last section concludes on the current extensions of this work.

2. Basic model

We only consider a synthesis subset of VHDL [IEE99] which excludes physical time and non-discrete types. We further require that processes be synchronized on a single clock edge, and be put in a normal form with a single `wait` statement at the beginning. Under these conditions, we may identify the simulation step with the clock cycle, and ignore clock signals.

Figure 1 shows an example design which computes the power function, producing $\texttt{ARG1}^{\texttt{ARG2}}$ in the output `RES`. Signal `START` begins the computation, and `DONE` indicates that the result is ready. This design is composed of two processes. Process `MULTIPLIER` computes `REG1` $\times$ `REG2`, it is activated by `REQ`, and it signals the result availability through `ACK`. Process `CONTROLLER` is a finite state machine that uses the functionality supplied by the first process.

According to the semantics of VHDL, the simulation of a design consists of repeating a step where each process is executed up to a `wait` statement, and all signals are then updated. A signal assignment x `<=` e evaluates expression e and generates a next value for x which takes effect in the following simulation step only. This behavior models wire delays. The ACL2 model of a VHDL design is a set of functions, macros and theorems describing its simulation behavior. Since ACL2 is an applicative framework, we follow Moore [Moo98] and represent each architecture as a transition function between states.

```
entity POWER is
  port (ARG1,ARG2: in NATURAL; START,CLK: in BIT;
        RES: out NATURAL; DONE: out BIT);
end POWER;
architecture BEHAV of POWER is
  signal STATE: NATURAL := 0;
  signal REG1,REG2,RESULT,COUNT: NATURAL;
  signal REQ,ACK: BIT;
begin
  MULTIPLIER: process begin
    wait until CLK = '1';
    if REQ = '1' then
       RESULT <= REG1*REG2;
    end if;
    ACK <= REQ;
  end process;
  CONTROLLER: process begin
    wait until CLK = '1';
    case STATE is
      when 0 =>
        if START = '1' then
           REG1 <= ARG1; REG2 <= 1; COUNT <= ARG2;
           STATE <= 1;
        end if;
      when 1 =>
        if COUNT = 0 then
           RES <= REG2; DONE <= '1'; STATE <= 3;
        else
           REQ <= '1'; STATE <= 2;
        end if;
      when 2 =>
        if ACK = '1' then
           REG2 <= RESULT; REQ <= '0';
           COUNT <= COUNT-1; STATE <= 1;
        end if;
      when 3 =>
        if START = '0' then
           DONE <= '0'; STATE <= 0;
        end if;
    end case;
  end process;
end BEHAV;
```

Figure 1: VHDL design for power function

2.1 States

The *state* of an architecture represents a snapshot of its variables, local signals, and interface signals (except the clock). It is implemented as a list of values as explained below.

- For each variable, there is one element holding the value stored in the variable.
- For each signal, there is one element holding the *current* value of this signal, and another element holding the *next* value of this signal. The next value represents the VHDL signal driver which holds only one value due to the restrictions on the time model imposed by the synthesis subset of VHDL.

By convention, the next value of signal x is named x+. For the previous example, the state is the list `(ARG1 ARG2 ... ARG1+ ARG2+ ...)`.

To separate the state behavior from its implementation, only two functions directly know about the actual state representation.

- Function (`getst` i st) fetches the value of variable i in state st. It returns the i-th element of state st
- Function (`putst` i a st) builds a new state and assigns value a to variable i. It returns a new state identical to st except for the i-th element, which then holds a.

For readability, we generate constants for indexes in the state list. A partial list from the previous design is shown below. Entity and architecture names prefix identifiers to avoid ambiguities in the ACL2 model.

```
(defconst *power.behav.arg1* 0)
(defconst *power.behav.arg2* 1)
...
(defconst *power.behav.arg1+* 12)
(defconst *power.behav.arg2+* 13)
...
```

The *characteristic properties* of `getst` and `putst` 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. Property P5 discharges an update when it assigns to a variable the same value it already stores.

P1: `(getst i (putst i a st)) = a`
P2: `i` $\neq$ `j` $\rightarrow$ `(getst i (putst j a st)) = (getst i st)`
P3: `(putst i a (putst i b st)) = (putst i a st))`
P4: `i` $\neq$ `j` $\rightarrow$ `(putst i a (putst j b st)) = (putst j b (putst i a st))`
P5: `(getst i st) = a` $\rightarrow$ `(putst i a st) = st`

From the properties above, ACL2 generates rewrite rules that reduce a nested expression (`putst` 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.

The naive application of the rewrite rule generated from property P4 could lead the ACL2 rewriter to an infinite loop, because the term it generates can be rewritten bay the same rule. ACL2 recognizes this possibility and employs a heuristic to stop such loops when the terms are ordered. However, the standard heuristic fails to identify which argument of `putst` must be used to stop the loop. Therefore, we add the clause below the theorem corresponding to P4:

```
:rule-classes ((:rewrite :loop-stopper ((i j))))
```

According this clause, the rule can only be applied when it swaps terms where `i` is larger than `j`.

The actual definitions of `getst` and `putst` are hidden after the proof of their characterizing properties, because the associated rewrite rules suffice to conduct all handling of states. Therefore the ACL2 model is independent of the state representation, and any other implementation that satisfies the same properties may be adopted, if it is found more convenient.

To describe a state in a formula, we use a *constructor expression* of nested `putst` calls beginning in `nil`, which, in this case stands for a predefined state where variables are undefined. For example, the formula below describes a state `st` where `ARG1` holds 15, `ARG2` holds 66, etc.

```
(equal st (putst *power.behav.arg1* 15
            (putst *power.behav.arg2* 66
               ...
                 nil)))
```

2.2 Transition Functions and their Uses

Using states explicitly, the execution of a group of VHDL statements in an initial state is represented as the expression that builds the corresponding final state. For instance, the execution of a signal assignment x `<=` e is modeled by (`putst` x e' *st*), which updates the next signal value, where e' corresponds to expression e with references to variables and signals replaced by the access to the corresponding elements in *st*. Using recursive function definitions and standard Lisp operators such as `cond` and `let*`, we model other VHDL statements.

Using this representation for statements, we model each process of the VHDL design by a *transition function* that returns the new state produced by *one simulation*

```
(defun power.behav.multiplier (st0)
  (let* ((st1 (if (equal (getst *power.behav.req* st0) 1)
                  (putst *power.behav.result+*
                    (* (getst *power.behav.reg1* st0)
                       (getst *power.behav.reg2* st0))
                    st0)
                  st0))
         (st2 (putst *power.behav.ack+*
                     (getst *power.behav.req* st1) st1)))
    st2))
(defun power.behav.controller (st0)
  ...)
(defun power.behav-update-signals (st)
  (putst *power.behav.res* (getst *power.behav.res+* st)
    (putst *power.behav.done* (getst *power.behav.done+* st)
      (putst *power.behav.state* (getst *power.behav.state+* st)
        ... st))))
(defun power.behav-cycle (st) (power.behav-update-signal
                                (power.behav.multiplier
                                  (power.behav.controller st))))
(defun power.behav-simul (n st)
  (if (and (intergerp n) (> n 0))
      (power.behav-cycle (power.behav-simul (- n 1) st)
      st))
```

Figure 2: ACL2 model for design of Figure 1

step of this process. For instance, in Figure 2, the functions `power.behav.multiplier` and `power.behav.controller` model the processes in Figure 1. One simulation cycle of architecture `behav` of entity `power` is modeled by function `power.behav-cycle` of Figure 2. This function calls the functions corresponding to each process, and then calls function `power.behav-update-signal` to replace the current value of each signal with its (possibly resolved) next value. More details about this modeling technique are found in [BGR00].

Finally, the *transition function* representing the *architecture* is a recursive function `power.behav-simul` that repeats the architecture simulation cycle n times, returning the final state. This ACL2 model may be *simulated* for n steps by evaluating the standard Lisp expression `(power.behav-simul` n st`)`, where st is a state with appropriate input values. It may also be used for *formal verification* by proving properties of function `power.behav-simul`.

For instance, the theorem below demonstrates that the VHDL design is correct. The hypothesis indicates that `st` is a state where the inputs `ARG1` and `ARG2` hold the natural numbers `m` and `n`, and the remaining variables have appropriate initial values. The conclusion indicates that the simulation beginning in this state arrives, after $3 \times \mathtt{n} + 2$ steps, in a state where `RES` holds $\mathtt{m}^{\mathtt{n}}$. This is a property of *all* simulation runs, for *all* positive values of `m` and `n`, covering an infinite number of cases. Model checkers cannot handle such general properties.

```
(defthm power.behav-correct
  (implies (and (integerp m) (>= m 0) (integerp n) (>= n 0)
                (equal st (putst *power.behav.arg1* m
                                 (putst *power.behav.arg2* n
                                   (putst *power.behav.start* 1
                                     ...
                                       nil)))))
           (equal (getst *power.behav.res*
                         (power.behav-simul (+ (* n 3) 2) st))
                  (power m n))))
```

The ACL2 model may also be *symbolically simulated*, which corresponds to executing the design with symbolic values as input. For instance, consider the goal below.

```
(implies (and (integerp m) (>= m 0) (integerp n) (>= n 5)
              (equal st (putst *power.behav.arg1* m
                               (putst *power.behav.arg2* n
                                 ...
                                   nil)))))
         (equal (power.behav-simul 10 st) v))
```

The hypothesis of this formula introduces a initial state `st` where the inputs are natural numbers `m` and `n`, with $n \geq 5$. When submitted to the symbolic simulator, the conclusion of this formula performs ten simulation steps beginning in `st`, which correspond to three iterations of the power algorithm. The result is the state below, where `REG2` contains the intermediate result $m \times m \times m$, as expected.

```
(putst *power.behav.arg1* m
  (putst *power.behav.arg2* n
    ... (putst *power.behav.reg2* (* m m m)
          (putst *power.behav.count* (+ n -3) ...))))
```

ACL2 manages to trace initial values and conditions along the whole run to decide tests over symbolic expressions. When it cannot decide a test, it performs a case split and explores all possibilities. More details about these verification and simulation techniques are found in [BGR00].

3. Components

The previous modeling technique needs to be extended to handle architectures with components. Figure 3 shows another design for the power function. In this example, the architecture `BEHAV` of entity `MULT` computes the multiplication of its input arguments when requested through signal `REQ`, and the architecture `COMP` of `POWER` uses it as a component. This design is equal to the design of Figure 1, except for the multiplication. In the new version, this operation is performed by a component and takes several simulation steps, while the first version performs it by a sequential process in one step. We now consider the modeling of components.

3.1 Component Representation and Verification

Architecture `BEHAV` of `MULT` has no component, so it is modeled as described before, with a state list of the form `(A B ... A+ B+ ...)`, and the corresponding transition functions, including the following two functions that compute one cycle (`mult.behav-cycle`), and the simulation of n cycles (mult.behav-simul) of this design.

```
(defun mult.behav-cycle (st) ...)
(defun mult.behav-simul (n st)
  (if (and (intergerp n) (> n 0))
      (mult.behav-cycle (mult.behav-simul (- n 1) st)
      st))
```

The state of each component of an architecture has internal variables and signals, and their values persist through simulation steps. Therefore, the state of an architecture with components includes *one state list* for each *instance of component*. For

```
entity MULT is
  port (A,B: in NATURAL; REQ,CLK: in BIT;
        PROD: out NATURAL; ACK: out BIT);
end MULT;
architecture BEHAV of MULT is
  signal STATE: NATURAL := 0;
  signal RESULT, COUNT: NATURAL;
begin
  MAIN: process begin
    wait until CLK = '1';
    case STATE is
      when 0 =>
        if REQ = '1' then COUNT <= A; RESULT <= 0; STATE <= 1; end if;
      when 1 =>
        if COUNT = 0 then
           PROD <= RESULT; ACK <= '1'; STATE <= 2;
        else
           RESULT <= RESULT+B; COUNT <= COUNT-1; end if;
      when 2 =>
        if REQ = '0' then ACK <= '0'; STATE <= 0; end if
    end case;
  end process;
end BEHAV;
architecture COMP of POWER is
  component MULT_UNIT
    port (A,B: in NATURAL; REQ,CLK: in BIT;
          PROD: out NATURAL; ACK: out BIT);
  end component;
  for all: MULT_UNIT use MULT(BEHAV);
  signal STATE: NATURAL := 0;
  ...
begin
  MULTIPLIER: MULT_UNIT (REG1, REG2, REQ, RESULT, ACK, CLK);
  CONTROLLER: process begin
    ...
  end process;
end COMP;
```

Figure 3: Power function with a component

example, the state for COMP of POWER is a list of the form (ARG1 ARG2 ... ARG1+ ARG2+ ... MULTIPLIER), where MULTIPLIER is a state for BEHAV of MULT, i.e., a list (A B ... A+ B+ ...).

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 transition function of the architecture bound to the component.

Figure 4 shows the ACL2 model for COMP of POWER. It is identical to the model in Figure 2, except for the function power.comp.multiplier which represents the component activation as follows:

1. It extracts its local component state st-mult0 (a state of BEHAV of MULT) from the global state st0.
2. Next, the component inputs A, B and REQ receive their actual value from the external architecture state, resulting in st-mult3.
3. A call to mult.behav-cycle activates the component internal processes, and produces state st-mult4.
4. The external architecture state is then updated with this new component state, resulting in st1 (a state of COMP of POWER).
5. Finally, the result is state st3, where the signals RESULT and ACK receive the component outputs.

Models of architectures with components can be used for execution, symbolic simulation and formal verification exactly as described in the previous section. In this last case, however, it is desirable that the proof effort be divided among the design components, and that properties proved over a component be carried to all designs using that component. However, how to achieve this goal is not obvious.

Consider again Figure 3. According to the approach discussed earlier, the theorem below for the multiplier correctness states that if a and b are the values of the input arguments, REQ is set to 1, and the internal signals are properly initialized, then the result found in PROD after the appropriate number of simulation cycles is a × b.

```
(defthm mult.behav-correct
  (implies (and ...
                (equal st (putst *mult.behav.a* a
                                 (putst *mult.behav.b* b
                                   ...
                                     nil))))
           (equal (getst *mult.behav.prod*
                         (mult.behav-simul ... st))
                  (* a b))))
```

```
(defun power.comp.multiplier (st0)
  (let* ((st-mult0 (getst *power.comp.multiplier* st0))
         (st-mult1 (putst *mult.behav.a*
                          (getst *power.comp.reg1* st0) st-mult0))
         (st-mult2 (putst *mult.behav.b*
                          (getst *power.comp.reg2* st0) st-mult1))
         (st-mult3 (putst *mult.behav.req*
                          (getst *power.comp.req* st0) st-mult2))
         (st-mult4 (mult.behav-cycle st-mult3))
         (st1 (putst *power.comp.multiplier* st-mult4 st0))
         (st2 (putst *power.comp.result+*
                     (getst *mult.behav.prod+* st-mult4) st1))
         (st3 (putst *power.comp.ack+*
                     (getst *mult.behav.ack+* st-mult4) st2)))
    st3))
(defun power.comp.controller (st0)
  ...)
(defun power.comp-update-signals (st)
  ...)
(defun power.comp-cycle (st) (power.comp-update-signal
                               (power.comp.multiplier
                                 (power.comp.controller st))))
(defun power.comp-simul (n st)
  (if (and (intergerp n) (> n 0))
      (power.comp-cycle (power.comp-simul (- n 1) st)
      st))
```

Figure 4: ACL2 model for design of figure 3

This is necessarily a property of function `mult.behav-simul`, because it takes several steps to compute the result. This function simulates an isolated multiplier, which does not interact with its surrounding environment. It does not consider, for instance, that the inputs may oscillate. So it is not appropriate as a description of the multiplier as a component.

Even considering only the multiplier properties which hold when it is employed as a component, it is not evident how they can be *lifted* from the component to the composite system. For instance, the correctness of the architecture `COMP` of `POWER` is stated as:

```
(defthm power.comp-correct
  (implies (and ...
                (equal st (putst *power.comp.arg1* m
                                 (putst *power.comp.arg2* n ... nil))))
           (equal (getst *power.comp.res*
                         (power.comp-simul ... st))
                  (power m n))))
```

This is a property over function `power.comp-simul`, which communicates with the multiplier component only through its step function `mult.behav-cycle`. But this is not the simulation function `mult.behav-simul` over which multiplier properties such as theorem `mult.behav-correct` are stated. Thus, properties of the multiplier component *cannot* be used in proofs about the composite architecture `COMP` of `POWER`.

3.2 Representation of Components Using Abstract Systems

The problems discusses above are consequences of the semantics of VHDL. To overcome them, we state and prove properties of an architecture $\mathcal{A}$ as if it were a component of an *abstract system* representing its *enclosing environment*. It stands for any other enclosing architecture $\mathcal{B}$ which uses architecture $\mathcal{A}$ the component, i.e., creates an instance of $\mathcal{A}$. The abstract system is not fully described, but only characterized as the simulation function of the enclosing architecture. This abstract system can later be specialized to any architecture, and the component properties are preserved in this process.

For instance, consider the description of architecture `BEHAV` of `MULT` as a component. A system using such component must obey two restrictions. First, the system state must include an element to hold the local state of the multiplier component. Second, each step of the system simulation function must activate the component through a call to function `mult.behav-cycle` on its local state.

Let st_m be the actual multiplier state. Let st_s be the state of an an unspecified, abstract system which has a multiplier as a component. To describe this abstract system, we employ the functions below.

- The abstract function (`mult.behav-getst` st_s) = st_m extracts the local state of the multiplier component from the enclosing abstract system state.

- The abstract function (`mult.behav-system` n st_s) = st'_s represents the system simulation function. This function repeats a simulation step which must perform the multiplier simulation step on the multiplier local memory.

In the case of the composite architecture `COMP` of `POWER`, these functions actually are (`getst *power.comp.multiplier*` st_s) and (`power.comp-simul` n st_s).

To fully characterize an architecture that includes the multiplier as a component, the two abstract functions above must correspond to functions with the following properties.

Initial state property: Executing zero step of the system simulation function does not change the multiplier state:

```
(equal (mult.behav-getst (mult.behav-system 0 st-s))
       (mult.behav-getst st-s)
```

Simulation step property: For any local and output element `x` of the component, one step of the system simulation function performs one simulation step of the multiplier component on its local state:

```
(equal (getst x (mult.behav-getst (mult.behav-system (1+ n) st-s)))
       (getst x (mult.behav-cycle
                  (mult.behav-getst (mult.behav-system n st-s)))))
```

This property must be stated for each local or output element `x` of the component.

The first argument of `equal` gets the value of this element after $n + 1$ system simulation steps. The second argument simulates the system n steps, performs one multiplier step on the local component state, and then extracts this element value. The property says that the result of both terms is the same.

Alternatively, we may say this property states that the paths in Figure 5 are equivalent (the diagram commutes).

The last property only holds for local and output elements of the component state because these are the only elements the component writes. The value of input elements of the component local state is taken from the global state as defined by the surrounding environment which writes to the component input ports in each simulation step.

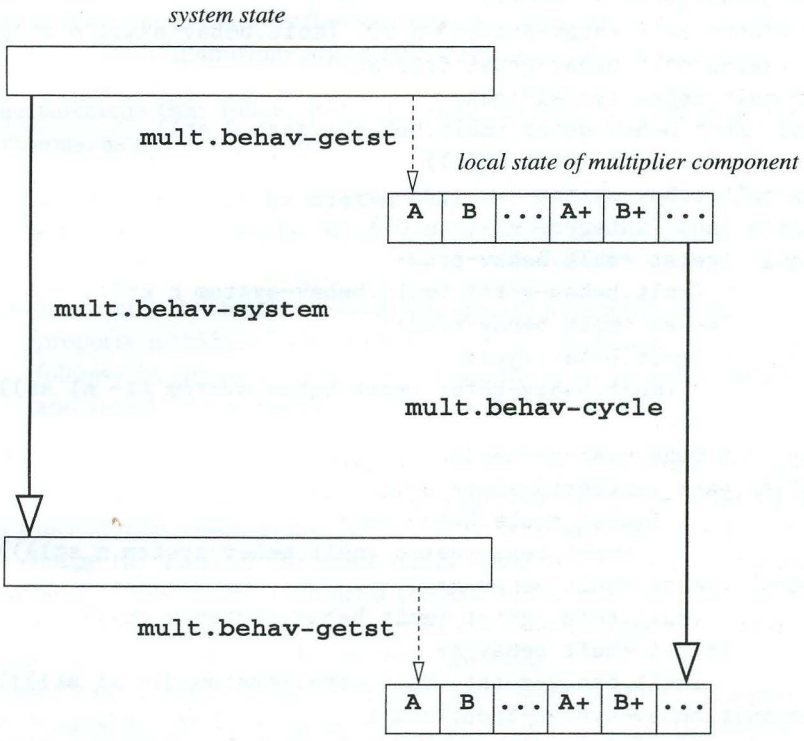

Figure 5: Simulation step of abstract function

```
(encapsulate
  (((mult.behav-system * *) => *)
   ((mult.behav-getst *) => *))
  (local (defun mult.behav-system (n st) (mult.behav-simul n st)))
  (local (defun mult.behav-getst (st) st))
  (defthm mult.behav-system-init
    (equal (mult.behav-getst (mult.behav-system 0 st))
           (mult.behav-getst st))))
  (defthm mult.behav-system-var-prod
    (implies (and (integerp n) (> n 0))
      (equal (getst *mult.behav-prod*
               (mult.behav-getst (mult.behav-system n st)))
             (getst *mult.behav-prod*
               (mult.behav-cycle
                 (mult.behav-getst (mult.behav-system (1- n) st)))))))
  ...
  (defthm mult.behav-var-a-stable
    (implies (and (integerp n) (> n 0)
                  (getst *mult.behav-req*
                    (mult.behav-getst (mult.behav-system n st))))
      (equal (getst *mult.behav-a*
               (mult.behav-getst (mult.behav-system n st)))
             (getst *mult.behav-a*
               (mult.behav-getst (mult.behav-system (1- n) st))))))
  (defthm mult.behav-system-induction t
    :rule-classes ((:induction :pattern (mult.behav-system n mem)
                               :condition t
                               :scheme (mult.behav-simul n mem)))))
```

Figure 6: Multiplier of design of figure 3 described as a component

In ACL2, the abstract functions `mult.behav-system` and `mult.behav-getst` may be introduced through the `encapsulate` construction, which introduces functions through their properties, without actually defining them. Figure 6 shows this construction.

- The second and third lines introduce the two abstract functions through their signatures.
- The next two lines give local witnesses definitions required by ACL2 to ensure that at least one actual function satisfies the constrains imposed to each abstract function. These definitions are immaterial in our presentation.
- The theorems that follow state the characteristic properties of the abstract functions, as discussed earlier.
  - Theorem `mult.behav-system-init` corresponds to the initial state property introduced above, which characterizes the beginning of the system computation.
  - Theorem `mult.behav-system-var-prod` corresponds to the simulation step property introduced above for the output component variable `PROD`. It is followed by similar theorems describing the other local and output variables and signals of the multiplier component.

Besides these properties characterizing the abstract functions, we may impose other restrictions to the abstract functions as additional theorems in the `encapsulate` body. In this example, theorem `mult.behav-var-a-stable` requires that the system does not change the value of the input signal `A` when there is a multiplier request. The hypothesis of this theorem checks if the `REQ` signal is active, and the conclusion indicates the value of `A` does not change in consecutive simulation states. Other restrictions may be imposed on the abstract functions.

The last theorem (`mult.behav-system-induction`) in the encapsulate is a fake property required by ACL2 to generate an induction rule for the abstract system function. It indicates we may perform induction on `mult.behav-system` using the same scheme of `mult.behav-simul`. This theorem is necessary because the definition of the system function is hidden, so its induction scheme is not visible outside the encapsulate.

After the abstract functions are introduced, the correctness of the multiplier can be stated through the theorem below. It follows from the characterizing properties of the abstract system which are exported (are not local) by the encapsulate, and the actual definition of the multiplier simulation step `mult.behav-cycle`.

```
(defthm mult.behav-system-correct
  (implies (and ...
```

```
              (equal (mult.behav-getst st)
                     (putst *mult.behav.a* a
                       (putst *mult.behav.b* b
                         ...
                           nil))))
        (equal (getst *mult.behav.prod*
                      (mult.behav-getst
                        (mult.behav-system ... st)))
               (* a b))))
```

Observe this is no longer a property of the component simulation function, but of its enclosing abstract system `mult.behav-system`. From now on, all multiplier properties are stated over this abstract system.

Using the `functional-instance` construct, this theorem can be later explicitly instantiated to any architecture which has the multiplier as a component. For instance, the theorem below instantiates it to architecture `COMP` of `POWER`.

```
(defthm power.comp-multiplier-correct
  (implies (and ...
              (equal (getst *power.comp.multiplier* st)
                     (putst *mult.behav.a* a
                       (putst *mult.behav.b* b
                         ....
                           nil))))
        (equal (getst *mult.behav.prod*
                      (getst *power.comp.multiplier*
                             (power.comp-simul ... st)))
               (* a b)))
  :hints
  (("Goal" :by
     (:functional-instance mult.behav-system-correct
       (mult.behav-system power.comp-simul)
       (mult.behav-getst
         (lambda (st) (getst *power.comp.multiplier* st)))))))
```

The `hint` construction in this theorem indicates it must be proved as an instance of the previously proved theorem `mult.behav-system-correct` substituting `power.comp-simul` for `mult.behav-system` and substituting `(mult.behav-getst st)` for `(getst *power.comp.multiplier* st)`. This proof proceeds as follows.

- First, we create an instance of the original theorem `mult.behav-system-correct` using the substitution above for `mult.behav-system-correct` and `mult.behav-getst`, and check that the proposed theorem `power.comp-multiplier-correct` is equivalent to this instance. This is a trivial step in the case above.

- Next, we check that the *substitution above satisfies to the restrictions imposed on the abstract functions* `mult.behav-system-correct` *and* `mult.behav-getst`. To perform this task:
  - we apply the same substitution above to all theorems on the body of the `encapsulate` that introduced these abstrac functions (Figure 6).
  - we prove each of these theorems.

  This step checks that the actual system `COMP` of `POWER` is an abtract system for the multiplier, i.e., it has `BEHAV` of `MULT` as a component.

The last step is the most time-consuming and it is repeated each time we create an instance of a theorem. However, we assume this step is reasonable easy, since it depends mostly on the structure of the functions involved. We must observe that the proof of the theorem being instantiated is not replayed. This is the greatest saving in this approach, since we expect the proofs of such theorems to be long and complex.

4. Conclusion

The long-term objective of this research is the formal validation of high-level specifications for digital systems being developed in an industrial context. Therefore we chose tools and methods which integrate smoothly with other design practices (such as value simulation), and aim at description techniques important for actual system developers.

This paper described the compositional verification of designs built from components. In our approach, we reason about the component embedded in an abstract environment, allowing for arbitrary restrictions on the environment and its components. This method can be extended to deal with systems where components are unspecified. Therefore, this method can support the modular development of designs, where third-party libraries and components play an essential role.

The work is still on-going and involves the automatic generation of properties for regular designs with generic parameters, and the application of our methodology to industrial designs.

References

[BGR00] D. Borrione, P. Georgelin, and V. Rodrigues. Symbolic simulation and verification of VHDL with ACL2. In *International Conference on HDL (HDL-CONF'2000)*, pages 167–182, San Jose, 2000.

[BM88] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[CGP00] Edmund M. Clarke Jr, Orna Grumberg, and Doron A. Peled. *Model Checking.* M.I.T. Press, 2000.

[CM88] K. Many Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, 1988.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL; A Theorem Proving Environment for Higher Order Logic.* Cambridge University, Cambridge, 1993.

[IEE94] IEEE. *IEEE Standard VHDL Language Reference Manual*, June 1994. Std 1076-1993.

[IEE99] IEEE Synthesis Interoperability W.G. 1076.6, New York. *IEEE Standard for VHDL Register Transfer Level Synthesis*, 1999.

[KM97] Matt Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–13, April 1997.

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer, 2000.

[McM93] K. C. McMillan. *Symbolic Model Checking.* Kluwer, Boston, 1993.

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

[Sha96] N. Shankar. PVS: Combining specification, proof checking and model checking. In *FMCAD'96*, pages 257–264, 1996. LNCS 1166.