Automatic generation of a provable circuit model: from VHDL to PVS

Katell Morin–Allory, Dominique Borrione

TIMA Laboratory, 46 avenue Félix Viallet 38031 Grenoble cedex France
katell.morin@imag.fr

This paper presents a method to automatically produce the formal model of a circuit design. Starting from a VHDL description, we generate a provable file in the PVS or ACL2 theorem prover. This method has been applied to the proof of correctness of monitors for a temporal property specification language.

■ 1. Introduction

The design of first time correct systems on a chip (SoC) involves, among many other challenges, guaranteeing that the system as designed will behave correctly. To face increasing time pressures, a design methodology called "platform–based design" is now popular in the European semiconductor industry: a new SoC is built according to a generic architecture, using pre–existing parameterized virtual modules and processor cores, the simulation models of which are retrieved and interconnected, possibly using some adaptation interfaces. Ideally, predesigned parameterized models should come with a formal proof of correctness for their register transfer level (RTL) implementation, and with a formal mathematical model allowing to reason about their usage in an embedding SoC. Industrial automatic tools, including model checking, SAT, BDD–based equivalence, etc., solve this problem for fixed size modules. In contrast, these methods cannot be used on parameterized virtual modules, which can possibly deal with unbounded size models and abstract data types. More elaborate techniques, such as the use of inference rules to establish properties on arithmetic and symbolic data, and inductive reasoning, are needed. Yet, one problem remains largely unsolved: ensuring that the mathematical model used in reasoning faithfully exhibits the semantics of its corresponding simulation model. This is the main topic of this paper.

We start from a RTL description, used for simulation and synthesis, written in the VHDL standard design language. A compiler and a simulator have been implemented in Mathematica by G. Al Sammane [Als05]. The symbolic simulation of a clocked synchronized sequential circuit is used to compute the state transition function and the output function of the underlying Mealy finite state machine, under the form of a normalized IF–THEN–ELSE expression. From that point, we automatically produce the mathematical formulation of the FSM [TBA04], either as recurrence equations, or as a set of step functions. Finally, the post–processing part of the tool produces the model in the input syntax of a theorem prover, for further reasoning on the model properties. Currently, we implemented the flow for ACL2 and for PVS.

The paper is structured as follows. In a first part, we give an overview of the overall system, and briefly describe its main modules. The second section concentrates on the modeling aspects: we present the recurrence equations and the step functions, and explain how they relate to the initial RTL design. The third section presents the main transformation algorithms, and our implementation in *Mathematica*. The last part illustrates the application of the tool to the automatic translation of a library of primitive modules into the input format of PVS, a necessary step prior to the library formal proof

of correctness, which has indeed been achieved. We conclude on the usefulness of this approach for proving that "what you verify is what you synthesize".

■ 2. Symbolic Simulation

Theosim [Als05] is a prototype symbolic simulation tool developed in our research team by Al Sammane *et al*. In a few words, it takes as input a clock synchronized sequential circuit, and computes the state transition functions and output function in a normalized conditional format. This tool performs a static stabilization of combinational circuits between clock edges. This simulator defines the symbolic value of a signal as a function of the previous symbolic value of all signals of its cone of influence.

The architecture of Theosim is given in Figure 1.
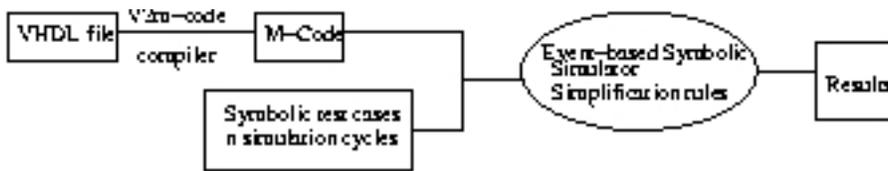
Input circuits The input circuit is described in the hardware description language VHDL. The principles of what follows would hold for Verilog as well. The supported VHDL (fully described in [Als04]) is compatible with the standard subset for synthesis [VHDL04], including the abstract data types, but it only recognizes the 1999 syntax for sequential processes.

Intermediate Format From the source VHDL file, a proprietary compiler extracts the defining expression of signals and variables for one simulation cycle (*i.e.* before any stabilization). It is implemented in *Mathematica*, a complete description is given in [Als04]. It is based on the extraction of the syntactic tree of the source VHDL which is turned into the Mathematica list format.

Symbolic Simulation The simulator is based on the VHDL simulation algorithm: at each simulation cycle, the active signals are updated, and processes sensitive to a modified signal are resumed. This step is repeated until stabilization of all signals: it is a delta cycle. Since the symbolic expression of signals and variables is simplified and normalized, a signal is stable if its symbolic value and its previous value are identical.

This simulation algorithm is used either to do symbolic simulation or numerical simulation. To perform a simulation for one clock cycle, one has to first set the clock edge, execute one clock synchronized simulation cycle, followed by one or more stabilization cycles. The resulting symbolic expressions are the results of the transition function for each state and output variable of the underlying finite state machine.

Modeling The transition functions given by Theosim express the symbolic value of a signal *s* at a given cycle in terms of its symbolic value at the previous cycle, and symbolic values of the other signals at the current or previous cycle. Two symbols are used for each signal *s*: the symbolic value at the current cycle is denoted *S* (capitalized letters of the signal name), the symbolic value at the previous cycle is denoted *S$*.

Example The VHDL text on Figure 2 is an excerpt from a primitive monitor RTL description that will be discussed in Section 4.

```
valid <= valid_t;
evaluate_expr: process(clk)
begin
  if clk'event and clk='1' then
```

```
   if reset_n='0' then valid_t <= '1';
   else if check_en = '1' then valid_t <= expr;
   else valid_t <= '1';
   end if; end if;
  end if;
 end process;
```

The output *Valid* is combinationally connected to an internal signal *Valid_t* (first line). The rest of the code is a sequential process that computes *Valid_t* at each rising edge of *clk*, as a function of signals *reset_n*, *check_en* and some expression *expr*. The *evaluate_expr* process and the assignment of *Valid* are concurrent, and require two simulation iterations to stabilize at each rising clock edge.

The symbolic simulation of signal *Valid* returns the following expression:
```
      VALID = if[RESET_N$ == 0, 1, if[CHECK_EN$ == 1, EXPR$, 1]]
```

Two equivalent models can be generated from the transition functions: a global simulation step function for the finite state machine, or individual functions for each state element of the machine. Which one should be preferred is application specific.

■ 3. Modeling functions

□ 3.1. Generation of a global simulation function

The first solution is to model the finite state machine by means of a recursive function.

Step function For each signal *X* we define a function *step* from the set of states (*i.e.* the set of value of all signals) to itself that computes the new state after one clock cycle. It consists in computing the image of each signal by the transition function.

$$\begin{aligned} step: \ &S \longrightarrow S \\ &s \mapsto s' \end{aligned}$$

Example Coming back to the previous example, let us denote D_{valid} (resp. D_{check_en}, D_{valid_t}, D_{Reset_t}, D_{expr}) the set of values of *valid* (resp. *check_en, Valid_t, Reset_t, expr*). The set of states *S* is defined by

$$S = D_{valid} \times D_{check\_en} \times D_{valid\_t} \times D_{Reset\_t} \times D_{expr}$$

and the *step* function is defined by:

$$\begin{aligned} step: \qquad\qquad &S \longrightarrow S \\ v, c, vt, r, e \mapsto\ &v', c', v', r', e' \end{aligned}$$

where *v'* is given by the transition function of *valid* (rewritten from *Valid_t*) (*i.e.* if *r=0* then *1* else if *c=1* then *e* else *1*), *c'* by the transition function of *check_en*, and *e'* and *r'* are the new values of input signals *expr* and *reset*.

Simulation function Then to compute the state after *n* cycles, a function *run* is defined.

$$\begin{aligned} run: \ &\mathbb{N} \times S \longrightarrow S \\ &n, s \mapsto \ \text{if } n = 0 \text{ then } s \\ &\qquad\qquad \text{else run } (n-1, step\ (s)) \end{aligned}$$

All signals are run concurrently.

☐ 3.2. Generation of individual signal functions

The other solution is to run signals independently. We could use a similar modeling technique, where the ''step'' and ''run'' functions would be defined for each signal, but this would be clumsy. Instead, we model signals by functions both in their structural and temporal dimensions, and we get recurrence equations. With this modeling, we can use different results on the formal verification of recurrence equation with affine dependencies especially when the dependencies are uniform and the domains of temporal and spatial indices have a polyhedral shape [CM04, BFR01].

In this model, the value of a signal *X* at the current cycle *t* is modeled by *X(t)* and its value one cycle before is modeled by *X(t−1)*.

As an example, Figure 3 gives the representation in PVS [SOR01] of the signal *Valid* defined in Figure 2.

```
VALID(t:nat): boolean =
(IF t=0 THEN TRUE
ELSE  IF NOT RESET_N_(t-1) THEN True
ELSE  IF CHECK_EN(t-1) THEN EXPR_(t-1)
ELSE True
ENDIF ENDIF ENDIF)
```

■ 4. Implementation

The transition functions given by Theosim are represented by their syntax tree and implemented by Mathematica lists. Our tool takes as input the list of signals and variables declared in the whole system, the desired representation (global or individual modeling of transition functions) and the language of a target theorem prover (PVS [SOR01] or ACL2 [KPM00]). The output is a PVS or ACL2 file specifying the system.

Our tool is composed of three main functions: `parseTree`, `sortSignals` and `fileBuilding`.

☐ 4.1. Parsing the syntax tree

Function `parseTree` turns the syntax tree of the transition function into PVS or ACL2 expressions. This transformation is done according to the selected representation:

Global modeling: Signals or variables are modeled by symbol; each one of their occurrences in the syntax tree is directly turned into a string.

I**ndividual modeling:** Signals or variables are modeled by time functions. Their occurrences are turned into function calls on time *t−1*.

□ 4.2. Signals sorting

In ACL2 and in PVS, an object A depending on an object B must be defined before B. For the individual modeling, signals need to be sorted according to their dependency order before being written into the output file. We construct a directed graph, where the nodes are the signal names and the edges represent the dependency between signals. This graph is implemented using *Mathematica* facilities. To sort our signals according to their dependency, we sort them in a topological order with `TopologicalSort`. This sort does not take into account the case of mutually dependent signals. We first detect them in the dependency graph (with `StronglyConnectedComponent`) then merge them into a new sub−node representing this set of mutually dependent signals. The topological sort is done on this new graph.

□ 4.3. Output File definition

The definition file is built according to the target language. For PVS, the preamble is automatically constructed (theory name, ...) and each symbol is typed. In ACL2 there is no preamble. Appendix A gives an excerpt of an automatically generated PVS file.

Global modeling: We generate the functions *step* and *run* defined in Section 3.1. according to the target language. Function `parseTree` is used to generate the PVS or ACL2 expressions.

Individual modeling: Each transition function is turned into a timed function, and written in the output file in the order given by `sortSignals`. For each inductive functions, the measure is automatically given (used to prove that the function is well defined), it is based on the decrease of time *t*. Mutually dependent signals are defined by a sub−function and a flag. In Appendix A, signals START_T1 and START_UNTIL are mutually dependent , they are defined by the sub−function START_REC. When the flag is 1, START_REC defines START_T1 otherwise it is START_UNTIL. For each signal the resulting timed function is initialized for *t=0:* signal initial values and types can be found automatically in the VHDL source syntactic tree. The type information is required in theorem provers like PVS, in ACL2 it is translated as a predicate on the signal domain and used as a precondition to enter a function body.

■ 5. Application

We developed an original method for generating hardware that monitors signals whose behavior is specified by logical and temporal properties under the form of assertions in the declarative form of the property Specification Language PSL. PSL is a standard specification language proposed by Accellera, and further standardized by IEEE [PSL]. It is based upon the Sugar 2.0 property specification language, and is an extension of the temporal logics LTL and CTL. PSL is used to describe properties that are required to hold in a device under verification (DUV).

We first briefly present the construction method and the library used to generate monitors. Then, we show how we use the method described in the previous section to automatically formalize the monitor into the logic of the PVS system, using *Mathematica*.

☐ 5.1. Monitor Construction

Our mechanism applies to any property specification language that ensures monotonic advancement of time, left to right through the property, such as the "simple subset" of PSL .

In the following, we use the VHDL flavor of PSL, but the principles apply to other syntax as well. In the text and the figures, 0 and 1 are used both for bits '0' and '1', and for Booleans False and True.

Monitors (*cf.* Fig 4) are built as modules that take as inputs the reset, the synchronization signals (clock, hand–shake, etc.) and the signals of the DUV that are operands of the properties to be monitored; their outputs code the level of satisfaction of the properties. As complex properties may be written by combining more elementary properties, corresponding complex monitors are built from the interconnection of more elementary monitors. We thus distinguish between internal operators/monitors that correspond to internal terms/sub–property; and last–level operators/monitors that correspond to the overall property. Monitors generated by this method have the same structure as the PSL formulae, and are thus easy to understand, to debug and to reuse. Both the "weak" and the "strong" version of the PSL operators are supported.
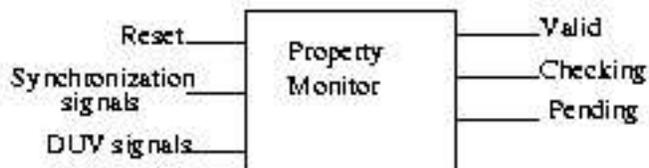

The monitor outputs have the following significance: *Checking* indicates when output *Valid* is effective and *Valid* provides the evaluation result (1 means absence of error, 0 means error). *Pending* means that the evaluation of the property has been started, but the result is still unknown; *Valid* is '1' when *Pending* is '1'. Outputs *Checking* and *Valid* are generated by the last monitor. They can be used to generate appropriate actions when a valuation of the property has been obtained.

Example: Assume we are interested in the behavior of signals *A, B, C* in a design synchronized by the rising edges of its master clock clk. We want to observe that signals *A, B, C* satisfy property *P*, expressed as:

```
Property P is always (A -> next_event![2](B) (C))
@rising_edge(clk);
```

P is an invariant (it must always hold), which states that whenever *A* is '1', it must be the case that the second times *B* takes the value '1', *C* takes the value '1' :
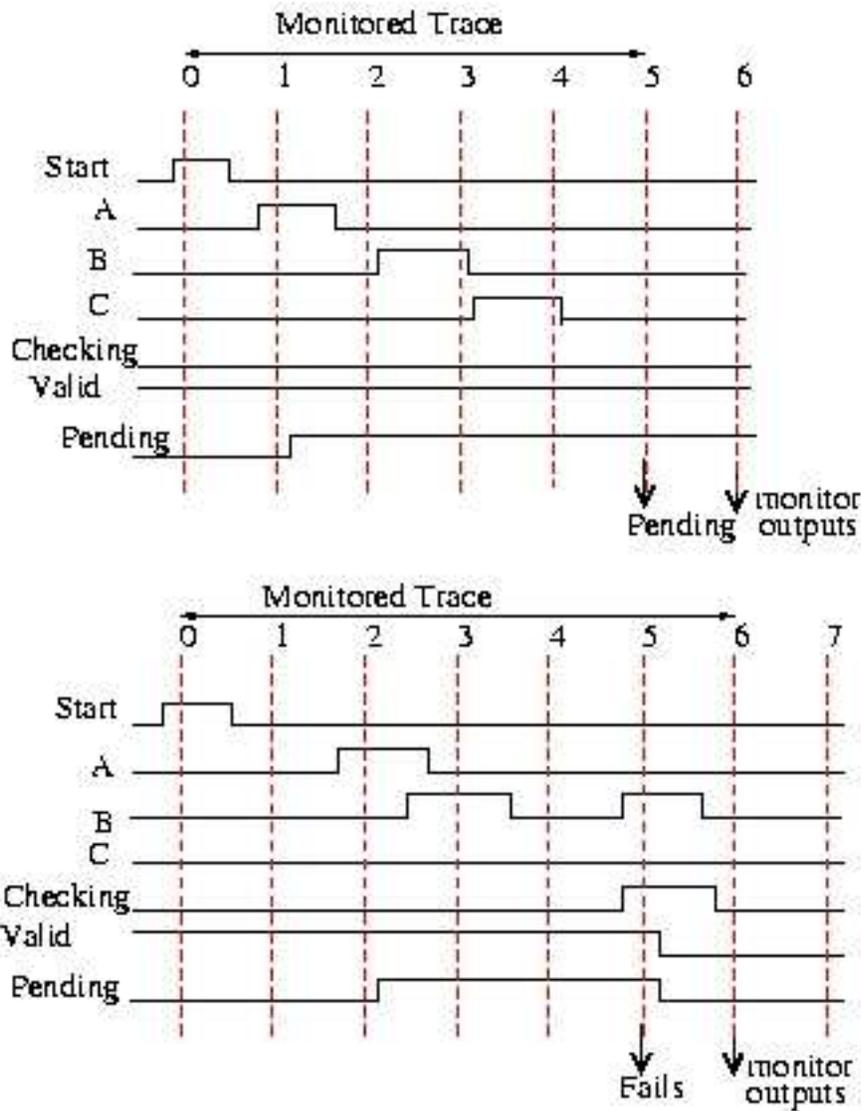
The waveforms on Figure 5 give two possible sequences of values observed on signals *A, B, C*. The vertical dotted lines represent the successive rising edges of *clk*, which have been numbered for the purpose of this explanation. *Start, Checking, Valid* and *Pending* are the input−outputs of the monitor for *P*, as explained above.

Top waveform: Property *P* is monitored on the trace from *clk* edge #0 to *clk* edge #5. At *clk* edge #1, *A* takes value '1'. Thus, starting from *clk* edge #1, property *P* holds on the trace ending at *clk* edge #5, if and only if *B* takes value '1' twice, and the second time *C* takes value '1'. *P* fails since *B* holds only once. At *clk* edge #6, the property is still pending. According to the design of our monitors, output *Pending* is significant only one cycle after the end of the trace.

Bottom waveform: Property *P* is monitored on the trace from clk edge #0 to clk edge #6. At clk edge #2, *A* takes value 1, and remains 1 for one clock cycle. On the waveform, *P* does not hold since the second time *B* takes value '1', *C* takes value '0' and *Valid* takes value '0' one cycle later. According to our design, output *Checking* takes value 1 at the same edge as *B* takes value '1' for the second time, and the value of output

Valid is significant one cycle later. Note that output *Pending* takes value '0', meaning that the property is not pending. If *C* took value '1' at clk edge #5, the value of *Valid* at *clk* #6 would take '1' and the property would hold.

Figure 6 shows the structural interconnection of primitive components to form the monitor for property *P*.
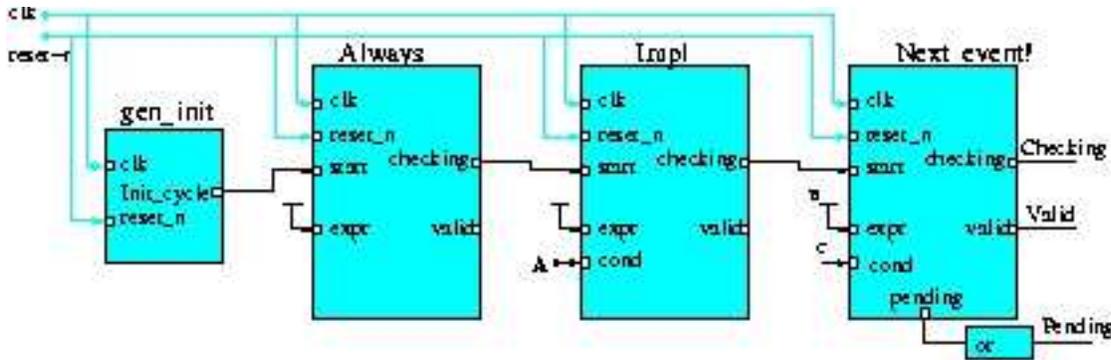
 The overall monitor takes as inputs the master clk and reset_n signals, and the observed signals *A, B, C*. It is the structural interconnection of primitive monitors, one for each temporal operator present in property *P: always, impl* (for '→'), *next_event.* An additional primitive module called *gen_init* starts the global monitoring process.

Structure of a primitive monitor:
The library monitors contains one primitive monitor for each FL operator of PSL. Operators that take one or two integer parameters, such as *next* or *next_a*, have corresponding generic monitors with the same parameters. All primitive monitors were designed according to a common basic structure of two main blocks. The first one, the *Checking Window Block* generates the temporal window for the evaluation of the operands, and sets an internal check enabling *check_en* signal, based on the evaluation requirement (*Start* input signal) and the semantics of the operator. The second one, *the Evaluation Block* executes the checking of the operands, when the checking enable signal is '1'. When *reset* is active, the monitor stays in its reset state. Otherwise, when *check_en* is active, the operand is checked and output *Valid* represents the result. When *check_en* is inactive, execution is stopped, and output *Valid* stays in its default value 1.

Construction of complex monitors by interconnection of primitive components :
To generate the monitors for complex properties, primitive single operator monitors are interconnected to construct a complex property monitor. The method is based on the syntax tree of the property. A node in the tree represents a PSL operator, a leaf represents a basic operand (signal or constant value), the edges connect an operator with its operands. Some operators have two operands, some have only one.

More details and example may be found in [BL05].

□ 5.2. Formalization

The proof of correctness of both the library components and the interconnection method, with respect to the formal semantics of the PSL reference manual was done with the system. The choice of PVS was motivated by the fact that the PSL semantics are expressed in second−order logic and thus directly represented by the input formalism. In addition, many proof strategies are automated.
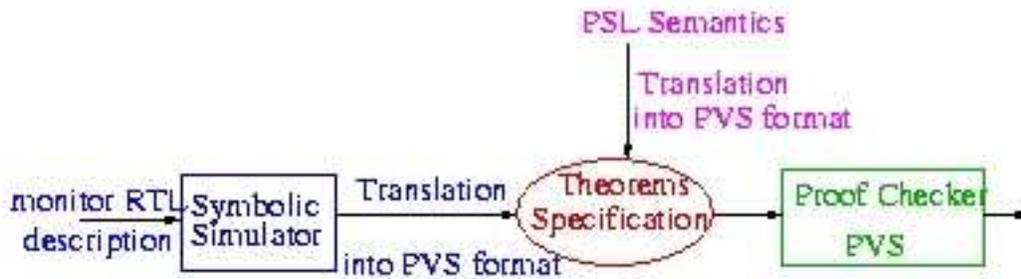
Monitors Modeling

The primitive monitors are written in the RTL synthesizable subset of VHDL. To prove each monitor correct, we extract its finite state machine (FSM) model, and translate it in the input formalism. This process has been automated, with the help of symbolic simulation (see Section 2). Appendix A gives the modeling file in for the monitor *until*.

Since we want to prove that the monitors implement the semantics if and only if some signals are active on a given number of cycles (a trace), we have chosen recurrence equations. Indeed, we are not interested on the whole state of our FSM (the monitor) at a given time but on the value of one signal on a whole trace. In this case, it is much easier to prove properties on signals modeled independently than concurrently. Furthermore, this modeling allowed us to benefit from the formal verification technique based on recurrence equations [QRMC06].

Then, the PSL semantics are represented in PVS, and we generate automatically several theorems for each operator. All theorems are proved with PVS and the proof can be found in [MB05].

■ 6. Conclusion

In this paper, we have described and illustrated a new method to verify the correctness of parameterized library components. The synthesizable design of each component, written inVHDL, is processed by a symbolic simulator that computes the transition functions of all the state and output variables of the design. Depending on the characteristics of the problem at hand, the designer may choose to reason in terms of recurrence equations on in terms of step functions, to prove his/her design compliant to a high level semantic specification.

A prototype tool has been implemented. The use of the rewriting and fixed−point computation of *Mathematica* has been crucial to quickly validate the overall approach and program the core algorithms of the symbolic simulator and provable model generators.

The main limitations to the dissemination of the tool are twofold. First, the user interface is currently very primitive, and requires prior knowledge of the *Mathematica* input syntax, which is acceptable for verification experts but not for hardware designers. Second, the complexity of the design that may be processed by the symbolic simulator is restricted by the memory space and allowed depth of nested expressions. As a consequence, our tool should be intended for use by expert formal verification engineers, who will perform, once and for all, the proof of correctness of library elements and modest size IP's to be used as building blocks for large projects.

■ Bibliography

[Als04] G. Al Sammane, *Specification of the VHDL subset supported by TheoSim,* Tima Laboratory, ISRN: TIMA−RR−−04/07−03−−FR.

[Als05] G. Al Sammane, *Simulation symbolique des circuits décrits au niveau algorithmique*, PhD thesis, Université Joseph Fourier, July, 2005, (in French).

[BL05] D. Borrione and M. Liu and P. Ostier and L. Fesquet, PSL−based online monitoring of digital systems, in Proc. *Forum on specification & Design Languages* (FDL'05), 2005.

[BFR01] D. Barthou and P. Feautrier and X. Redon, *On the Equivalence of Two Systems of Affine Recurrence Equations*, RR n° 4285, INRIA, 2001.

[CM04] D. Cachera and K. Morin−Allory, Verification of safety properties for parameterized regular systems, in *Trans. on Embedded Computing Sys.*, 4:2, 2005, ACM Press .

[KPM00] M. Kaufmann and P. Manolios and J Strother Moore, in Proc. *ACL2 Computer Aided Reasoning: An Approach*, Kluwer Academic Press, 2000.

[MB05] K. Morin−Allory and D. Borrione, A proof of correctness for the construction of property monitors, in Proc. *High Level Design Validation and Test (HLDVT'05)*, Dec., 2005, IEEE Press.

[QRMC06] P. Quinton and T. Risset and K. Morin−Allory and D. Cachera, Designing Parallel Programs and Integrated Circuit, in Proc. *International Mathematica Symposium*, Avignon, 2006.

[PSL] *1850 IEEE Standard for Property Specification Language*, IEEE, October, 2005.

[SOR01] N. Shankar and S. Owre and J.M. Rushby and D.W.J. Stringer−Calvert, *Prover Guide*, 2001, Computer Science Laboratory, SRI International.

[TBA04] D. Toma and D. Borrione and G. Al Sammane, Combining Several Paradigms for Circuit Validation and Verification, in Proc. *CASSIS 2004*, 2005, vol. 3362, LNCS, Springer.

[VHDL04] *1076.6−2004 IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*, IEEE, 2004.

■ **Appendix A**

```
mnt_until_monitor : THEORY

BEGIN

% ASSUMING
 % assuming declarations
weak_excl_op_1 :  nat=0;
strong_excl_op_1 :  nat=1;
weak_incl_op_1 :  nat=2;
strong_incl_op_1 :  nat=3;
op_type_1 :  nat =weak_excl_op_1;
% ENDASSUMING

% VHDL DEFINITIONS

PENDING_T(t:nat):RECURSIVE boolean=
(IF t=0
THEN FALSE
ELSE IF (RESET_N(t-1) = False)
THEN False
ELSE IF (COND1(t-1) = True)
THEN False
ELSE IF (START(t-1) = True)
THEN True
ELSE PENDING_T(t-1)
ENDIF
ENDIF
ENDIF ENDIF)
MEASURE t

START_REC(flag:subrange(1,2),t:nat):RECURSIVE boolean=
IF flag=1 THEN
(...)
ELSE
(...)
ENDIF
MEASURE t

START_T1(t:nat):boolean=START_REC(1,t);
START_UNTIL(t:nat):boolean=START_REC(2,t);

PENDING(t:nat):RECURSIVE boolean=
(...)
MEASURE t

CHECKING(t:nat): boolean =
(...)

VALID(t:nat): boolean =
```

```
(  ...)

% END VHDL DEFINITIONS

% THEOREMS
(to complete)
% END THEOREMS

END mnt_until_monitor
```