proc. of CHARME'95 , Frankfurt, Germany , oct.95, LNCS 987.

A Symbolic Relation for a Subset of VHDL'87 Descriptions and its Application to Symbolic Model Checking

Emmanuelle Encrenaz

Laboratoire MASI / IBP,
Université Pierre et Marie Curie,
4 place Jussieu,
75252 Paris Cedex 05, FRANCE

e-mail : Emmanuelle.Encrenaz@masi.ibp.fr

Abstract

This paper presents the main principles for building a symbolic transition system from a description written in a subset of VHDL'87 (temporal information is excluded and objects are restricted to bit, bit_vector and Boolean types). This transition system is used for formal verification of the VHDL description. It consist of a system of Boolean equations indicating the next state of the system in terms of its current state. It is automatically generated from an intermediate representation of the VHDL description by means of a Petri Net. The deterministic nature of VHDL 87 and the exclusion of temporal elements in the description permit us to abstract the behavior of the system : only one state per delta cycle is represented instead of all intermediate states encountered in simulation. This abstraction reduces the size of the transition system and the cost of subsequent analysis. The construction of the system of Boolean equations from the Petri Net is presented first, and then an example of verification of a temporal logic property illustrates its use for Symbolic Model Checking. Experimental results are given which demonstrate the feasibility of this approach.

1. Introduction

The increasing complexity of hardware systems makes their design verification difficult by classical simulation techniques. Formal verification techniques have appeared as a complementary verification method of hardware systems. Their links with VHDL began at the end of the 80's, and since much research has been done in this area. Different formal semantics of VHDL'87 [VHDL'87] were proposed to apply automatic formal verification : theorem proving techniques [VanTassel93] [BPS92], stream functions [BS-FD-K94], process algebra [BSCPB94], structural Petri Nets analysis [OC93], Symbolic Model Checking [DB93] DB95] [DH95a]...

Symbolic Model Checking techniques require the description of the behavior of the system by means of a transition system determining the states of the system and the relations between them. Recent trends of symbolic representation of transition systems by means of BDDs [Bryant86] have established the viability of symbolic Model Checking for real hardware systems [BCMDH90] [McMillan93]. Our aim is to further the link between VHDL descriptions and Symbolic Model Checking techniques. This requires the definition of a formal model representing the VHDL system, from which a symbolic transition system is extracted. Once this transition system is obtained, classical Symbolic Model Checking algorithms can be applied [McMillan93].

Building the transition system characterizing the behavior of the system is a difficult task : the building mechanism is simple but the explicit representation of states and relations induces a combinatorial explosion. The time needed to build the explicit transition system and the space to store it become prohibitive. A symbolic representation of the transition system and a symbolic simulation engine overcome these complexities. This approach is well suited to non deterministic systems (such as VHDL'93 where the use of shared variables introduces non determinism in the simulation cycle).

The deterministic nature of VHDL'87 can be exploited to directly obtain the symbolic transition system from the structure of the formal model without symbolically simulating it. This approach is the one adopted by [DH95b] who build a detailed symbolic transition relation representing the behavior of the VHDL system in all the intermediate states of the simulation. Their approach is applicable to a fairly large subset of VHDL (all finite types and timing constructs are allowed in their descriptions). [DB95] also define a semantics, based on abstract machines, that represents the behavior of a subset of VHDL (timing constructs are excluded but all finite types are treated). The symbolic transition system built in [DB95] is coarser than the one in [DH95b] as the only states considered are the ones at the end of each delta cycle.

We present a way to obtain a symbolic transition system from a restricted VHDL description (timing constructs are excluded and objects are of bit, bit-vector and Boolean types). The transition relation is expressed as a system of Boolean equations, each of which is built from the structure of an intermediate Petri Net. The level of abstraction is similar to [DB95] : only states at the end of each delta cycles are considered. We illustrate the use of the symbolic transition system by presenting a means of detecting never-ending simulation cycles in a VHDL description. With present simulation tools, these are difficult to detect and to correct. Experimental results of symbolic model checking show the relevance and feasibility of our approach.

This paper is organized as follows : Section 1 briefly presents the structure of the Petri Net derived from a VHDL description and shows that, in our case, the state at the end of a delta cycle depends only on the state at the end of the previous delta cycle and the configuration of stimuli applied. Thus, the behavior of the system can be abstracted to end-of-delta states only. Section 2 presents the major rules for building a symbolic transition system from an intermediate Petri Net. Section 3 presents an application of this system of Boolean equations to symbolic model checking. The requirement that a VHDL simulation stabilizes is represented as a temporal logic formula to be verified. Verification of this property is presented on a scalable example with experimental results. Section 4 concludes and suggests some directions for future work.

1. A Formal Model for a VHDL Description

VHDL semantics is informaly expressed by means of its simulation engine. One has to develop a formal model to reason about a VHDL description. We chose the Petri Net formalism as it supports non determinism that will be necessary for VHDL'93, but it can also represent deterministic systems, such as VHDL'87. Various techniques of construction of the transition system representing all behaviors of the modeled system can be applied to this formalism. An overview of Petri Nets can be found in [Murata89].

A VHDL description is automatically translated into a Petri Net [BEC94]. The Petri Net represents the control structure of VHDL processes and their synchronization reproducing the VHDL simulation semantics. An external data part of the Petri Net contains the data modified by the firing of transitions in the control part. The construction and behavior of the Petri Net are presented in [EB95]. They are briefly reviewed in the following section.

1.1. Petri Net Features

The Petri Net is composed of subnets, reproducing the structure of each VHDL process and their synchronization according to VHDL'87 semantics. Each process is composed of places and transitions. Places refer to the states of the process and transitions are fired to pass from one state to the next, representing the VHDL statement executed between these two states. Connections between places and transitions are expressed by Pre and Post matrices. Pre(t,p) = 1 indicates an arc from p to t, and Post(t,p) = 1 indicates an arc from t to p. Transitions modeling processes are split into two disjoint sets. Those modeling VHDL *wait* statements belong to the RES set, they are only firable during the resumption phase of VHDL delta cycles. All other VHDL statements are represented by EXE transitions, which are firable during the execution phase of VHDL delta cycles.

A global scheduler emulates the delta cycle functioning of VHDL simulation by decomposing the delta cycle into RESUME, EXECUTE and UPDATE phases which provide synchronization barriers for the processes.

This Petri Net interacts with an external data part that represents the VHDL data objects. We consider variables, constant and signals : effective, driven, resolved, driver connection values and signal attributes (event and transaction). These objects are restricted to bit, bit_vector and Boolean types.

Interactions between the control part and the data part occur while transitions are fired. These interactions are represented by means of attributes associated to each transition, t, of the Petri Net :

- g(t) is the guard of transition t : t may fire only if its guard is true. g(t) is a Boolean function of data contained in the data part.
- ASG(t) is the set of data modified while firing transition t.
- TRF(t) is the set of transformations applied to the data in ASG(t). TRF(t) is a set of couples $(d,trf_{d,t})$ where $d \in$ ASG(t) and $trf_{d,t}$ is a Boolean function of data in the data part.

In VHDL'87, all statements enclosed between two *wait* statements in a process are atomic : instead of being represented by a sequence of EXE transitions, each corresponding to a VHDL statement, they can be represented by a unique EXE transition grouping all the data transformations.

We characterize $SQ(t_{init},t_{out})$, a sequence of EXE transitions between two RES transitions of a given process :

$SQ(t_{init},t_{out}) = t_{init} \rightarrow t_a \rightarrow t_b \rightarrow t_c \rightarrow ...t_n \rightarrow t_{out}$

where $t_{init} \in$ RES, $t_{out} \in$ RES, and $t_a,..., t_n \in$ EXE.

This sequence can be reduced to $SQ(t_{init},t_{out})_{reduced} = t_{init} \rightarrow t_{initout} \rightarrow t_{out}$

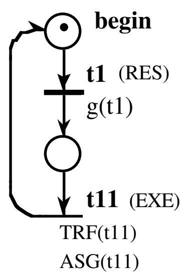
where $t_{initout} \in$ EXE, and

- $ASG(t_{initout}) = ASG(t_a) \cup ASG(t_b) \cup ... \cup ASG(t_n)$
- $g(t_{initout}) = g(t_a) \wedge g(t_b) ... \wedge g(t_n)$
- $TRF(t_{initout}) = TRF(t_a) \blacktriangleright TRF(t_b) \blacktriangleright ... \blacktriangleright TRF(t_n)$

  "$\blacktriangleright$" means "followed by" : the data modifications of t_a are combined appropriately with those of t_b, etc...

Reduction rules are defined in [EB95]. The example on Figure 1 illustrates the Petri Net structure and the data formalism introduced here.

```
entity add_1b is
port ( a0,b0,ci0 : in bit;
       s0,co0: out bit )
end add_1;
architecture example of add_1b is
begin
add_0 : process
   begin
   wait on a0,b0,ci0              -- t1
   s0 <= a0 xor b0 xor ci0;       -- t2
   co0 <= (a0 and b0) or          -- t3
   (a0 and ci0) or (b0 and ci0);
end process add_0;
end example;
```



DATA = {(a0,eff_a0,drv_a0,evt_a0), (b0,eff_b0,drv_b0,evt_b0), (ci0,eff_ci0,drv_ci0,evt_ci0), (s0,eff_s0,drv_s0,evt_s0), (co0,eff_co0,drv_co0,evt_co0)}

ASG(t1) = Ø   ASG(t11) = {drv_s0,drv_co0}   TRF(t1) = Ø

TRF(t11) = {(drv_s0,eff_a0 xor eff_b0 xor eff_ci0), (drv_ct0,(eff_a0 and eff_b0) or (eff_a0 and eff_ci0) or (eff_b0 and eff_ci0)}

g(t1) = evt_a0 or evt_b0 or evt_ci0   g(t11) = TRUE

Fig. 1. Petri Net representation, interacting data part (DATA) and attributes derived from a VHDL description.

t1 belongs to the RES set as it represents the *wait on* statement, and *t11* represents the sequence of statements executed during an EXECUTE phase. It results from the merging of statements t2 and t3.

1.2. Behavior of the Petri Net

The behavior of the Petri Net reproduces the sequences of states of the VHDL program encountered during a simulation. In particular stable states (stabilization of the VHDL simulation cycle) and non-stable simulation cycles are represented.

A state S of the Petri Net is defined as : S = < M , D >, where

- M is a marking of the Petri Net : a configuration of tokens in places of the Petri Net, indicating the current state of each process and of the scheduler.
- D is a data configuration : the value of each data element in the interacting data part.

The structure of the reachability graph of the Petri Net exhibits the deterministic nature of delta-cycles in VHDL'87 semantics : *the state reached at the end of an execute phase is entirely determined by the stimuli applied to the system and the state, at the beginning of this execute phase.*

This final state depends exclusively on the following elements before the resumption of processes :

- The state of each process,
- The value of readable internal or output data (current value of variables and effective value of internal or output signals),
- The value of some attributes (event of internal or output signals),
- The configuration of stimuli applied to the system (their effective values and their event attributes).

In the general case, the update of effective values and the generation of events depend on the sequence of transactions in drivers and on the previous effective value of the signal. This implies a global clock that time stamps all transactions in all drivers and maintains their consistency.

If we assume that each driver contains one unique future value, and updates of the effective values of signals are performed during the update phase following their assignment, no global clock is needed : the effective values and events of internal and output signals depend only on the execution during the previous delta cycle. This assumption restricts the VHDL signal assignment : the clause *after* is not allowed. The absence of a global time excludes the use of *wait for* statement.

Thus, we are able to determine a state at the end of an *execute* phase from the state at the end of the previous *execute* phase. The evolution of such a system may be represented as a set of Boolean equations representing the next state of the system from the knowledge of the current state. Such a system of Boolean equations can serve as the basis for further analysis of systems, such as Symbolic Model Checking or Automata Equivalence.

This relation between end-of-delta states hides all intermediate states encountered during the delta cycle. This reduction of the number of states is beneficial for analyses that follows : the transition system is represented with a smaller number of variables and the number of iterations of Symbolic Model Checking algorithms is reduced : micro-steps inside a VHDL delta cycle are replaced by a unique step representing the whole delta cycle.

The next section presents the rules for building the system of Boolean equations from the Petri Net, and this is followed by an example of its applicability to Symbolic Model Checking.

2. Expression of a State in Terms of the Previous One

This section describes the behavior of the Petri Net as a set of Boolean equations. This representation of the transition system, characterizing the deterministic behavior of the Petri Net, is directly extracted from the structure of the Petri Net, avoiding an -often long and costly in space!- reachability graph construction. This representation is applicable to deterministic systems (the Boolean equations are deterministic by nature), and the representation of one state per delta cycle is applicable under the assumption that events in an *update* phase come either from changes in the stimuli or from changes in the internal or output data generated during the previous *execute* phase. It can be applied to VHDL'87 descriptions from which temporal clauses are excluded.

2.1. Location of Boolean Variables in the Delta Cycle

2.1.1. Set of Boolean Variables

From the previous remarks, a state S = <M,D> at the end of the *execute* phase of a delta cycle can be expressed in terms of the state at the end of the *execute* phase of the previous delta cycle.

The Boolean variables that define a state S are :

- Places P_k of the Petri Net preceding RES transitions (when not in the *execute* phase, all processes are on a *wait* statement, hence the only potentially marked places are the preconditions of RES transitions). Places P_k represent breakpoints in processes.
- Information concerning each signal sig :
  - effective value : eff_sig,
  - driven values of internal or output signals : drv_sig, (in case of multiple drivers : drv_pi_sig)
  - event (transactions if necessary) : evt_sig (trs_sig) ,
  - driver connection of internal or output resolved signals : connex_drv_pi_sig,
- Variables : var_k,

Let $\mathcal{V}$ be the set of these Boolean variables. $\mathcal{V}$ can be split into disjoint sets, each representing a particular category of Boolean variables : $\mathcal{P}$ represents breakpoints in processes, *eff* the effective values of signals, *drv* (or *drv_pi*) the driven values, *evt* the events, *connex_drv_pi* the connection of driver, and *var* the VHDL variables.

The value of all variables at instant *i* can be expressed as a function of the value of the variables at instant *i-1*.

$$\forall\, v_k \in \mathcal{V},\ v_k[i] = f_k(v[i-1] \in \mathcal{V})$$

f_k is a Boolean function extracted from the structure of the Petri Net and the guard and data transformations associated with the transitions.

2.1.2. Locating the State Relative to the Delta Cycle

Each simulation cycle is composed of an *update* phase followed by an *execute* phase. The future state of a system, S[i+1], can be expressed in terms of the current state S[i].

We could locate S[i] at the end of the *ith* delta cycle, in other words between the *execute* and *update* phases. In this representation, events on signals need not be represented : they are computed from driven and effective variables in each equation where an event is needed.

Another location of S[i] is at the end of the *update* phase of the *i+1th* delta cycle. In this case, for a non resolved signal, driven and effective values are equal. Hence, the driven values of non resolved signals need not enter into the functions.

Both approaches for S[i] induce similar systems of equations. However, the location of S[i] between *update* and *execute* phases seems better for systems that do not not contain resolved signals : event generation is computed only once. With the location between *execute* and *update* phases, event generation is computed in every equation where it is necessary. In the case of systems containing resolved signals, the number of Boolean variables in the location of S[i] between the *execute* and *update* phases is smaller. Figure 2 shows the Boolean variables defining S[i] between the *update* and *execute* phases. The notation introduced in Figure 2 are used throughout the paper.

We distinguish external variables, representing stimuli, from internal and output variables. Internal or output variables have their behavior constrained by the system, and their evolution can be expressed by predicates. External variables have no predictable evolution, and they appear as non constrained variables.

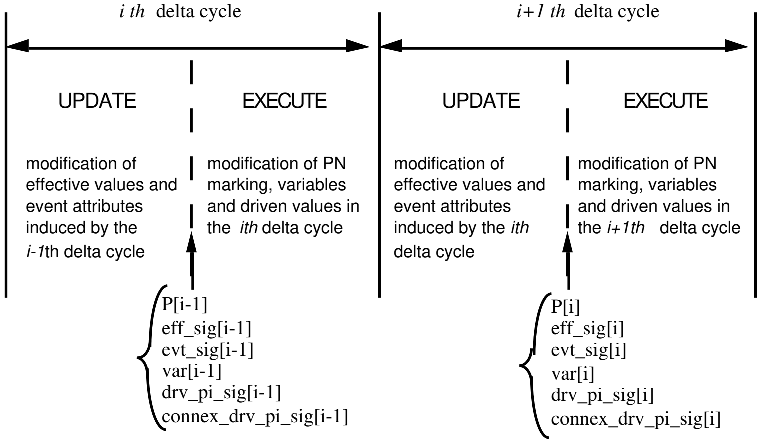


Fig. 2. Location of Boolean variables in the VHDL delta cycles.

Marking of places in the Petri Net and values of data at a given instant can be expressed in terms of the same variables at the previous instant. Boolean variables that represent driven values of signals and connections of drivers are only needed for resolved signals.

2.2. Equations of Boolean variables representing a state S

Four classes of Boolean functions are introduced. They represent the expression of the six types of variables defined in section 2.1.1. at instant $i+1$ in terms of their values at instant i.

The construction of the equation of each variable follows a simple reasoning : We try to answer the question *"what are the conditions the Petri Net must meet at instant i, for a place (resp. a datum) to be marked (resp. to be true) at instant i+1?"*. These conditions are translated into a Boolean formula that becomes the right hand side of the equation characterizing the evolution of the place or datum considered.

For conciseness, we will only consider the case of non resolved signals. In the following four sections, we present the method for generating the Boolean equations of variables of type $\mathcal{P}$, *eff*, *var* and *evt*.

2.2.1. Equations for Preconditions of RES Transitions

The task for determining the equation for the precondition of a RES transition presented and illustrated on an example.

Let P_k be a place Precondition of a RES transition t_k. t_k represents a breakpoint of the process and the marking of P_k indicates if the process is waiting in the *wait* statement indicated by t_k. By construction of the Petri Net, there exists at least one sequence of VHDL execution statements leading to the *wait* statement represented by t_k.

The fact that a place P_k is marked in S[i+1] results from two possibilities:

- P_k was already marked in S[i], and either the token didn't leave P_k, or it did leave P_k and it returned to P_k, or
- P_k was not marked in S[i], but the execution sequence which was fired during the delta cycle leads to the marking of P_k.

Figure 3 presents a general configuration of P_k and illustrates the construction of its equation.

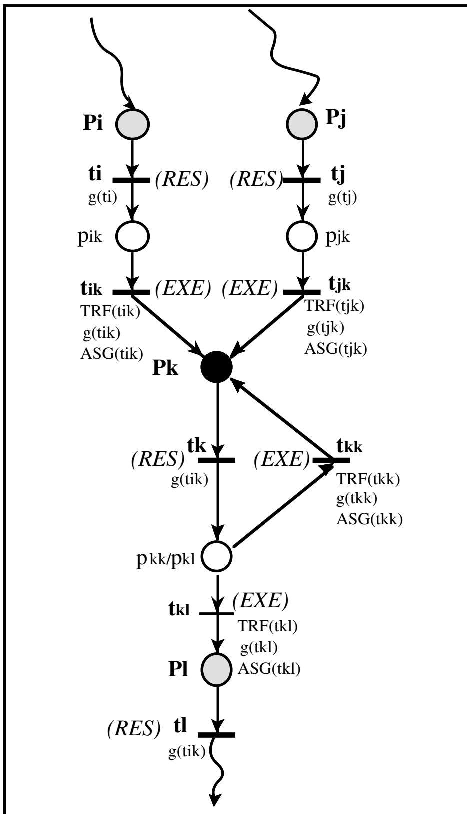
The marking of place P_k in S[i+1] can be represented as a predicate reproducing the alternatives detailed above :

$$\boxed{P_k[i+1] = (P_k[i] \wedge \neg (g(t_k)[i])) \vee (\bigvee_{SQ} (P_{init}[i] \wedge g(t_{init})[i]) \wedge g(t_{initout})[i]))} \quad (1)$$

where $SQ = SQ(t_{init},t_{out})_{reduced} = t_{init} \rightarrow t_{initout} \rightarrow t_{out}$, s.t. $Pre(P_{init},t_{init}) = 1$, and $t_{out} = t_k$

The first part of the right hand side of the equation corresponds to the previous marking of P_k, and the second part corresponds to a disjunction of all sequences terminating in t_k, with a given initiator P_{init}.

The guard of a RES or EXE transition is represented by a Boolean function of *var* , *eff* and *evt* variables.



Place P_k is precondition of a RES transition t_k. t_k is the output transition of sequences $SQ(t_i,t_k)$ and $SQ(t_j,t_k)$, and the input transition of sequences $SQ(t_k,t_k)$ and $SQ(t_k,t_l)$.

P_k is marked in S[i+1] if :

- P_k was marked in S[i] and transition t_k was not firable

$$P_k[i] \wedge \neg\ (g(t_k)[i])$$

- or, P_k was marked in S[i], t_k was firable and $SQ(t_k,t_k)$ was the sequence fired during the i+1th delta cycle

$$P_k[i] \wedge (g(t_k)[i] \wedge g(t_{kk}[i]))$$

- or, P_k was not marked in S[i] but the execution sequence fired during the delta cycle leads to the marking of P_k. This means that P_i (resp. P_j) was marked, and the corresponding sequence was fired : t_i (resp. t_j) was firable and the data values were such that $g(t_{ik})$ (resp. $g(t_{jk})$) was firable.

$$(P_i[i] \wedge g(t_i)[i] \wedge g(t_{ik}[i])) \vee (P_j[i] \wedge g(t_j)[i] \wedge g(t_{jk}[i]))$$

Fig. 3. Expression of the marking of place P_k (in black) at instant i+1 in terms of markings of other places (in grey) and firability of RES transitions at instant i.

2.2.2. Equations for the Effective Value of an Internal or Output Signal

With the signal assignment restriction in mind, the effective value of an internal or output signal in S[i+1] is not modified if no sequence assigning a value to this signal was fired during the previous delta cycle. If a sequence modified at least one of the signal's drivers, the effective value of the signal is updated with a consistent value (the driven value in the case of non resolved signals, and the resolved value in the case of resolved signals). In the particular case of non resolved signals, no driven value variable is needed because of the definition of S[i]. There may be more than one assignment to the given signal in the process pi; this is expressed as a disjunction of all sequences modifying the driver in pi. At a given instant, at most one sequence in the process pi is firable.

$$\boxed{\begin{array}{l} eff\_sig[i+1] = (eff\_sig[i] \wedge \neg\ (\bigvee_{SQ \subset pi} P_{init}[i] \wedge g(t_{init})[i] \wedge g(t_{initout})[i])) \\ \quad \vee (\bigvee_{SQ \subset pi} trf_{drv\_sig,t_{initout}} \wedge P_{init}[i] \wedge g(t_{init})[i] \wedge g(t_{initout})[i]\ ) \end{array}} \quad (2)$$

where $SQ = SQ(t_{init},t_{out})_{reduced}$ and P_{init} is a place in process pi, such that

$Pre(t_{init},P_{init}) = 1$, and $drv\_sig \in ASG(t_{initout})$.

The first part of the right hand side of the equation stipulates that no sequence modifying drv_sig was fired, and the second part corresponds to the firing of one sequence modifying drv_sig. In this last case the new value assigned to the driver, $trf_{drv\_sig,tinitout}$, is the new value for eff_sig after the update phase. $trf_{drv\_sig,tinitout}$ is a Boolean function of *var*, *eff* and *evt* variables representing the assignment of the driven value of signal sig in the sequence SQ.

2.2.3. Equations for the Value of a Variable

The equation for a variable var_k is similar to that for the effective value of a non resolved signal, except that "eff_sig" is replaced by "var_k" in equation 2.

2.2.4. Equations for the Value of an Event of an Internal or Output Signal

With our assignment restriction (no *after* clauses allowed in signal assignment), the occurrence of an event on an internal or output signal in S[i+1] implies that one of the processes assigning this signal has resumed in the previous delta cycle, and the assigned value was different from the effective value.

In the case of non-resolved signals, the expression of an event is straightforward : it is an exclusive OR of its driven value at the end of the i+1th delta cycle and its previous effective value.

$$evt\_sig[i+1] = drv\_sig[i+1] \oplus eff\_sig[i]$$

The driven and effective values are equal for non resolved signals, thus drv_sig[i+1] = eff_sig[i+1]. The expression of evt_sig[i+1] can be rewritten as :

$$\boxed{evt\_sig[i+1] = \bigvee_{SQ \subset pi} (P_{init}[i] \wedge g(t_{initout})[i] \wedge (trf_{drv\_sig,t_{initout}} \oplus eff\_sig[i])),} \quad (3)$$

where $SQ = SQ(t_{init},t_{out})_{reduced}$ and P_{init} is a place in process pi, such that
$Pre(t_{init},P_{init}) = 1$, and $drv\_sig \in ASG(t_{initout})$.

The right hand side of the equation indicates that an event occurs only if a sequence modifying the driven value was fired, and this driven value, represented by $trf_{drv\_sig,tinitout}$, is different from the effective value of the signal. The disjunction considers all sequences in process pi modifying the driver of the given signal.

2.3. System of Boolean Equations

We have shown in section 2.2. that each Boolean variable of a state S[i+1] can be expressed as a predicate of the variables in S[i]. This model is applicable to VHDL descriptions where temporal information is excluded.

The size of the transition system is known from the VHDL description : if signals are non resolved, a system containing S stimuli, I internal signals, O output signals, V variables and W wait statements will induce a system of (W + 2(I + O) + V)

Boolean equations with 2(W + 2(S + I+ O) + V) Boolean variables. Each Boolean variable is duplicated in order to distinguish the current state from the next one. This is an upper bound : we may not represent the event variable of non assigned signals, or the $\mathcal{P}$ variables of processes containing a unique *wait* statement.

In the case of resolved signals, two sets of variables are added as they are necessary to express event and effective values : the driven values for each signal and for each assigning process, and the driver connections for each driver of each resolved signal.

This system of equations represents the functional transition system of the VHDL program. The transition relation is expressed as R = {vj[i+1] = fj(v1[i],...vn[i]) }, and Symbolic Model Checking algorithms can be applied [McMillan93]. As an example, we propose to study the stability of simulation cycles under the assumption of stability of stimuli.

3. Example : Characterizing the Stability of a Simulation Cycle

3.1. VHDL Simulation and Stabilization Points.

A VHDL simulation is composed of simulation phases taking place at real-time instants; themselves are composed of a sequence of delta cycles taking no real-time to execute. A simulation phase ends when a stable state is reached : only a real time increment will force the system to evolve, thus inducing a new simulation phase.

It may happen that a simulation phase never ends, this means that the system never encounters a stable state in a given simulation cycle. Two kinds of non stabilization can occur : a process never ends its execution (*never-ending delta cycle*) or two or more processes resume each other in the same simulation phase (*never-ending simulation phase).*

These non stabilizations are difficult to detect and to correct. They are generally detected while a simulation is performed : a simulation cycle never stops and as a consequence, expected results are not obtained. The detection of a never-ending delta cycle is simpler as one has only to insure that each loop in each process terminates. Never-ending simulation phases may imply more than one process, and the absence of simulation results make the detection of dependency cycles difficult. The only prevention from infinite loops proposed by some simulators is the imposition of a maximum number of delta cycles per simulation phase. When this maximum number of delta cycles is reached in a given simulation phase, the simulation halts.

Classical examples of such non stabilizing systems are RS flip-flops or memory elements with a bad initialization. For elements known to be unstable, an *assert* VHDL statement can be used to warn of a bad configuration of signal values when it occurs in simulation. This prevention is only possible when the user is aware of non stable elements, and the verification is limited to the simulation. The difficulty comes from the fact that a system may present non stabilizations which are not anticipated.

Synchronous designs adopting strict description rules avoid this problem [DJ92]. But in a general case, [VHDL'87] does not prevent from these never-ending loops.

Conditions for never ending simulation cycle, provided stimuli do not change during the cycle, are of two types : the system structure and signal values have to be taken into account. The system must contain cyclic processes dependencies with no timing clauses. This condition however is not sufficient. In some cases, a cylic processes dependency will never oscillate. In other cases, it will oscillate if certain signal values are present. And then there are also cases where it will oscillate regardless of the signal values.

The next section proposes the use of Symbolic Model Checking techniques to characterize systems presenting these never-ending simulation cycles.

3.2. Detection of non Stabilizing Simulation Phases using the System of Boolean Equations.

The system of Boolean equations developed in section 2 can help detect never-ending simulations. The stabilization of a simulation cycle corresponds to a termination property that can be expressed in CTL and checked by Symbolic Model Checking techniques [McMillan93] :

Let T be the termination property to be verified : T is an atomic property that is true in the terminal states. The detection of sequences that always stabilize comes down to the computation of the CTL formula **AF**(T). This computation returns all states from which all outgoing paths contain a state where T is true. The set of states inducing a never-ending simulation cycle are the result of the negation of **AF**(T) : **EG**(¬T).

In the system of Boolean equations, a stable state is characterized by

$$T = (\neg \bigvee_{sig \in I \cup O} evt\_sig)$$

Thus the characterization of never-ending VHDL simulations comes down to the computation of

$$\mathbf{EG}(\bigvee_{sig \in I \cup O} evt\_sig)$$

where **EG**(p) is the greatest fixed point of the iterative equation :

$$iter[i+1] = iter[i] \bigwedge \exists v' \, (R \wedge iter[i]_{(v \text{ <- } v')})$$

iter[i] is the result of the equation at the ith iteration, R is the symbolic transition relation (it is a conjunction of all Boolean equations obtained as presented in section 2.), ∃v' is the existential quantification of primed variables, and (v <-v') indicates that ordinary variables are replaced by primed variables in the boolean function.

If the intersection of the set of bad states with the set of reachable states is non empty, the resulting set contains the set of reachable states from which a non stabilization occurs.

3.3. Example : a bus arbiter with combinatorial loops

The example used for demonstrating our approach is composed of *n* cells of a bus arbiter proposed by [Mc Millan 93]. A VHDL description of this device is described in [Clarke94].

The VHDL description of one cell of the bus arbiter is composed of three processes. The interconnection of cells induces cyclic dependencies which may drive the system into a never-ending simulation.

Questions we want to answer are : *Does such a system present never-ending simulation phases under the assumption that stimuli do not change during the simulation phase ? If so, can we characterize the states that induce this non stability ?*

The CTL property to verify is : $\mathbf{EG}(\bigvee_{sig \in I} evt\_sig)$ which returns all states from which some outgoing paths will never stabilize.

Experimental Results

All experiments have been performed on a Sun SPARCStation 10 with 32Mbytes of memory. We used the BDD package supplied by [Long] to manipulate Boolean functions.

nbr of cells	nbr of (current) variables : total (wait,evt,eff)	Equation system (CPU time in s)	Image Computation (CPU time in s)	EG Computation (CPU time in s)
2	44 (7,18,19)	0.1	6.5	1.3
3	63 (10,26,27)	0.2	36.9	3.0
4	80 (13,30,37)	0.3	158	6.7
5	104 (16,42,46)	0.5	323	15.7
6	120 (19,50,51)	0.7	2709	150

The set of states inducing never-stabilization is characterized. Its intersection with the set of reachable states returns an empty set : the non stabilizing sequences are not reachable from the initial state. Other initilisations may have induce non stabilization. The performance results show that :

1) The computation of the system of Boolean equations from the Petri Net does not take very much time and grows quite linearly with the size of the system.

2) The longuest part of the time is spent during the image computation. Once this image built, the computation of the bad states takes a few seconds because the intermediate results from an iterative equation of the EG computation are constrainted by the image set.

4. Conclusion and Future Work

In this paper, we have presented a method to extract a symbolic transition system from a restricted VHDL description to automate Symbolic Model Checking. The symbolic transition system is expressed as a system of Boolean equations that characterizes the behavior of deterministic systems. The system of Boolean equations is derived from the structure of an intermediate Petri Net formalism. All intermediate states encountered in a VHDL simulation are not represented : only one state per delta cycle is taken into account. This grouping of states in a delta cycle speeds up the analysis methods by abstracting the behavior of the system. This abstraction is only possible for VHDL descriptions without temporal clauses. Major principles of the construction of the system of Boolean equations from the Petri Net are presented in this paper. This Boolean system can then be used for Symbolic Model Checking or Automata Equivalence. An example of Temporal Logic Property verification is given : the stabilization of VHDL simulation cycles under the stability of stimuli is expressed as the termination property of a system and verified by Symbolic Model Checking algorithms. Experimental results scaling up to a system of 100 variables show the applicability of such an approach.

It is our intention to extend this approach to the verification of classical temporal logic properties on larger systems. Future directions include :

1) The evaluation of state encoding : the state encoding presented here is not fully compact : if a process contains several (say q) *wait* statements, each *wait* statement will be encoded on a Boolean variable. As each process is an automaton, the state of each process could be represented with $\log_2 q$ Boolean variables. The drawbacks of this compact representation are the management of non relevant combinations and the resulting increase in complexity of the Boolean equations. Tests will have to be performed to evaluate the benefits of a compact encoding of states.

2) The evaluation of Symbolic Model Checking algorithms on the system of Boolean equations without building the image of the system. This implies a caracterisation of the behaviour of stimuli, that may only change at stabilisation points. The equations of these stimuli will be added to the relation.

3) The study of good heuristics for variable ordering in BDDs by examining the variable dependencies between the left and right parts of equations. An intuitive approach consists of pairing each current variable with its next state representation. This could be extended to other dependencies.

Bibliography

[BCMDH90] J.R.Burch, E.M.Clarke, K.L. McMillan, D.L.Dill, L.H.Hwang, *"Symbolic Model Checking : 10^{20} states and beyond"*, Proc. 5th IEEE Symposium on Logic in Computer Science, 1990, pp 428-439.

[BEC94] R.K.Bawa, E. Encrenaz, J.M. Couvreur, *"VPN Technical Report"*, technical report IBP-MASI 94-13, Apr.94.

[BPS92] D.Borrione, L.Pierre, A.Salem,*"PREVAIL: A proof environment for VHDL descriptions"*,in Correct Hardware Design Methodologies, ed P.Camurati and P.Prinetto, North Holland 1992

[Bryant86] R.E.Bryant, *"Graph Based Algorithms for Boolean Function Manipulation"*, Transaction on Computers, Vol C-35, pp. 677-691, 1986.

[CBM90] O. Coudert, C. Berthet, J-C. Madre, *"Verification of sequential machines using functional vectors"*, in International Workshop on Applied Formal

Methods for Correct VLSI Design, volume VLSI Design Methods II, pp. 179-196, Belgium 1989. IFIP WG 10.2/ WG 10.5, North Holland 1990.

[Clarke94] E. Clarke, "A VHDL subset for Model Checking", CMU Internal Report, feb. 95.

[DJ92] A. Debreil, D. Jaillet,*"Synchronous description in VHDL for formal proof and resulting guidelines proposed by BULL"*, Advanced Report, BULL Produits et Systèmes, Dpt Développements Assistés, Les Clayes sous Bois, France, Jul 1992. BULL/92.0001 rev.A.

[BS-FD-K94] P.T.Breuer,L.Sanchez-Fernandez, C.Delgado-Kloos, *"Proof Therory and a Validation Condition Generator for VHDL"*, Proc of the EURO-DAC,Grenoble, 1994, pp 512-517.

[BSCPB94] C.Bayol, B.Soulas, F.Corno, P.Prinetto, D.Borrione,*"A Process Algebra Interpretation of a Verification Oriented Overlanguage of VHDL"* .Proc of the EURO-VHDL, Grenoble France 1994, pp. 506-511.

[DB93] D.Déharbe,D.Borrione,*"Symbolic Model Checking of VHDL Design Entities"*, Technical Report of IMAG Institut, RR 925 -I, dec 1993

[DB95] D.Déharbe,D.Borrione,*"A qualitative finite subset of VHDL and semantics"*, Technical Report of IMAG Institut, RR 943 -I, feb 1995

[Döhmen94] G. Döhmen, *"Petri Nets as Intermediate Representation between VHDL and Symbolic Transition Systems"*, Proc of the EURO-VHDL, Grenoble France 1994, pp. 572-577.

[DH95a] G.Döhmen, R.Herrmann,*"A Deterministic Finite-State Model for VHDL"*, Formal Semantics for VHDL, edited by C. Delgado Kloos and P.T. Breur, Universitad Politecnica de Madrid, Spain, feb 1995.

[DH95b] G.Döhmen, R Herrmann, *"Translating VHDL into functional symbolic finite-state models"*, special issue of Formal Methods In System Design, D.Borrione Editor, Kluwer Academic Publisher. To appear in 1995.

[ECB93] E. Encrenaz, J-M. Couvreur, R-K. Bawa, *"Validation of VHDL systems based on Petri Net modeling"*, in : Proc Workshop on Design Methodologies for Microelectronics and Signal Processing, Poland, 1993

[EB95] E. Encrenaz, R-K. Bawa, *"A Petri Net Model for Verifying Properties of VHDL programs"*, Research Report MASI-IBP 95-07, Feb 95.

[Long] D.E. Long, "A Binary Decision Diagram Package", manual page.

[McMillan93] K. Mc Millan, *"Symbolic Model Checking"*, Kluwer Academic Publisher, Norwell Massachusetts, 1993

[Murata89] T.Murata, *"Petri Nets : Properties, Analysis and Applications"*. Proc IEEE, vol 77 n°4, apr 89, pp 541-580.

[OC93] Olcoz - Colom , *"A Petri Net Approach for the Analysis of VHDL Descriptions"*, in: Proc of the CHARME 1993.

[TSLBS-V90] H.J.Touati, H. Savoj, B. Lin, R.V. Brayton, A. Sangiovanni-Vincentelli,"Implicite State enumeration using bdd's",Repot, University of California, Berkeley, USA, 1990.

[vanTassel93] J.P. Van Tassel, *"Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assitant"*, PhD Thesis , University of Cambridge, 1993.

[VHDL'87] "*IEEE Standard VHDL Language Reference Manual*" IEEE Std 1076-1987.