

A Petri Net Model of VHDL: A Base to Apply Formal Methods to Hardware Design

Serafín Olcoz

Design Technology Department. TGI, S.A.

Velázquez 134-bis. E28006 Madrid. Spain..

Phone: (34) 1-3964911. Fax: (34) 1-3964841. E-mail: serafin.olcoz@sp1.y-net.es

Abstract.

There is a need for formal verification techniques in hardware system design. This requires a formal interpretation of VHDL is the language most used by designers to describe and synthesize electronic systems. The formal model proposed is based on Coloured Petri Nets and can cover all aspects of the language. The starting point is the interactive processes executable model of the language. The formal model of a description includes the specification in Petri Net terms of a model consisting of the user-defined processes resulting from the elaboration of a VHDL description, the kernel process (simulator) and the communicating links between them.

1. Introduction.

The standard VHDL (IEEE-STD.1076) [1], has become the most widely supported Hardware Description Language (HDL) by industry, research centers and CAD vendors. This HDL covers a wide range of description levels (from system level to gate level) and supports different description styles (structural, data-flow and behavioral). A new design methodology based on VHDL is emerging, supported by different tools covering most aspects of design cycle of VLSI circuits (simulation, synthesis, etc.). Since a standard HDL is used by many users with different cultures, there are a need for a method of sharing a detailed interpretation on the language. Reference, [2] shows the consequences of a non formally defined HDL.

Moreover new release of the standard, [3], does not solve these problems. Furthermore, when VHDL descriptions are used for other purposes other than these considered in the VHDL Language Reference Manual (LRM, [1]), then the user and/or the tool vendor defines an application dependent semantic of the language (e.g., for synthesis purposes). These semantics usually define a subset of VHDL for the particular application. The semantics defined for these subsets may not be compatible with the standard definition of the language, and often these new semantics are based on a poor understanding of the LRM.

In order to help to any possible user, from hardware designers to VHDL-based tool implementors, this paper tries to clarify the language simulation semantic presenting a Coloured Petri Net (CPN) model of the VHDL execution. Although this formal model has been defined following the first release of the LRM, [1], it can be easily extended in order to capture the changes introduced by the new LRM, [3]. Other works on semantical models can be found in [10-14].

The paper is organized as follows. Section 2 describes the discrete event-driven simulation. Section 3 defines the execution model of the language. Section 4 presents CPN representation of variables, data types and expressions appearing in VHDL descriptions. Section 5 presents the CPN representation of the VHDL statements, subprograms and processes. Section 6 shows the architecture of the tools needed to automatically obtain a formal of any VHDL description. Finally, Section 7 presents the conclusions of this work.

2. VHDL Event-Driven Simulation.

VHDL and other HDLs can be considered a branch of programming High Level Languages (HLLs). The specific purpose of HDLs, to describe hardware, results a different processing than with general purpose HLLs.

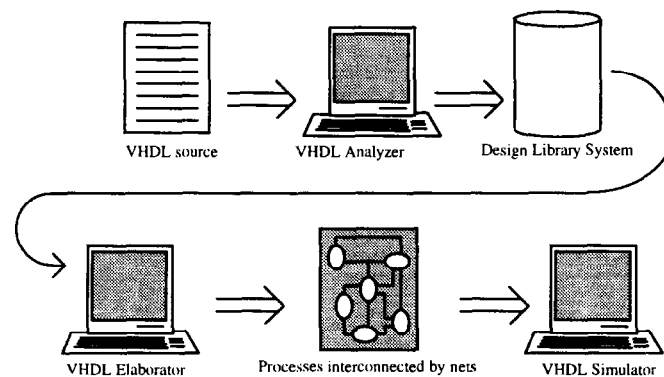


Figure 1.- Analysis, Elaboration and Execution of a VHDL description.

To execute a HLL source code it must be first analyzed, and then it must be compiled, the program has to be translated into the native language of the host computer.

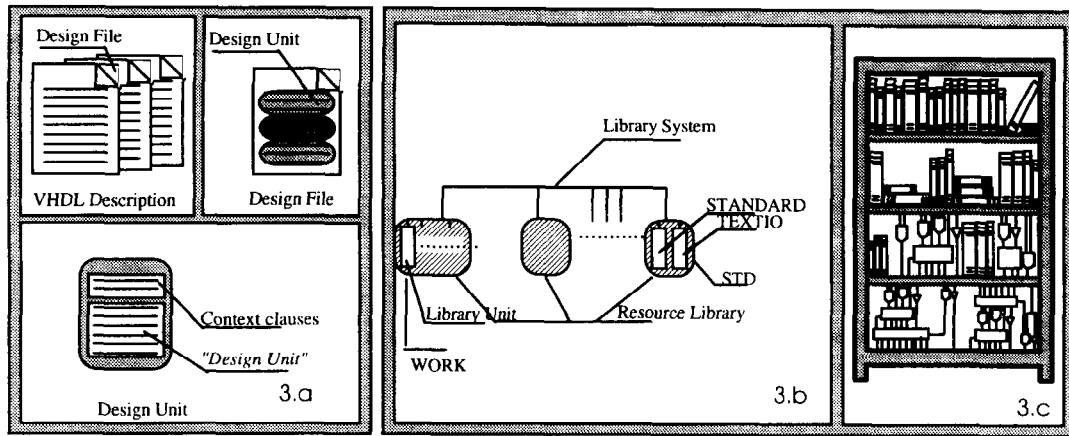


Figure 2.- Analysis, Elaboration and Execution of a VHDL description.

The execution of an HDL description also starts with the analysis, but before the translating to the native machine code we need to process the analysed results to generate an executable model. The execution of this model simulates the behavior of the system described by means of the HDL.

The underlying executable model of the language is a set of interactive processes. One of these processes corresponds to an event-driven simulator that manages time and schedules events. The other processes represent the behavior of the system described by the designer. The interactive processes paradigm represents discrete-event dynamic systems. Therefore, a simulatable HDL can be considered as the programming language of a discrete event simulator, that represents a particular class of dynamic systems, the hardware systems.

Figure 1 shows the processing stages we need to perform before the description can be simulated, [4].

First, the design files containing the source code must be analyzed to produce a design library that allows management and reusability of the design information.

Some constructs of the language, such us *USE* context

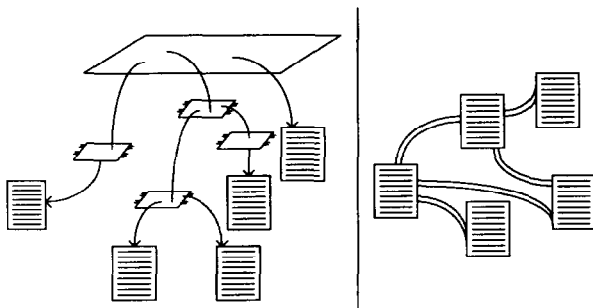


Figure 3.- Elaboration of a VHDL description:

Transformation of a design hierarchy into an heterarchy of VHDL processes interconnected by VHDL signals.

clauses, disappear during the analysis of a design file and their effect consist of creating the relationships between the library units forming the design library, see

figure 2. In fact, this part of the language semantics can be considered as corresponding to the interface language of a database whose structure is based on the entity-relationship model. This feature of the language is not formalized in this work.

The language analyzer, however, does not create an executable model. The VHDL executable model is generated by the elaborator, which transforms a design hierarchy into an heterarchy of VHDL processes interconnected by a network of VHDL signals, and their associated information, see figure 3. In a heterarchy of processes there are no processes inside other processes but the model is not flat because processes can use subprograms (functions and procedures). Besides to transforming a network of some library units in the design library into an executable model, the elaboration of the language declarations creates the objects defined by the declarations used in the execution of the model.

The elaborated model does not depend on the description style (behavioral, structural, data-flow, or mixed), chosen by the designer, or on the abstraction level of the VHDL description. This model contains the same information that the original description but is described using only a subset of the language which corresponds with the commonly defined as "behavioral" VHDL, the rest of the constructs disappear during their elaboration. The correspondence between this behavioral model and the original description can be recovered by means of the backtracking information produced by the elaborator. The semantics aspects of VHDL elaboration are not formalized in this work.

Finally, the elaborated processes are asynchronously and concurrently executed by the simulator.

The elaborated processes together with the designer's information about the window of simulation time (between 0 fs and Time'High) are the inputs of the simulator. Elaborated processes are composed by VHDL sequential statements, that are cyclically executed. Each process has its own state. Both, variable and signal

values can be modified by means of their corresponding assignment statements. Each process has also its own control that can be modified by means of: conditional statements (if and case); iterative statements (loops, both for and while in conjunction with next and exit); and wait statements. When a process executes a wait statement it suspends execution until the conditions of the statement are satisfied and then, resumes. A process can also contain assert statements, useful for simulation purposes, but without any influence in the model behavior.

Each elaborated process is sensitive to events on signals; in other words, its execution is event-driven. Events arrive to a process through the signals and the process execution may also produce events on signals that are sent to the processes' network. The way in which projected events are queued depends on the timing model used by the signal assignment statements of the processes. Inertial and transport delay models are the two preemptive delay models available in VHDL. Both models have a different algorithm of adding future events to the signal's driver, [1].

The heterarchy of processes resulting from the elaboration of any VHDL description could be considered as the interface language of a discrete event-driven simulator. This simulator is also defined by the LRM and manages the time advance, the activity of the processes during the simulation, and the updating of the signals.

3. The VHDL Execution Model.

The VHDL simulation semantics has been defined through the elaboration of an executable model, by an abstract event-driven simulator, [1]. The execution of the VHDL elaborated processes by a simulator creates a software model whose evolution simulates the behavior of the hardware system described by the VHDL

description. This software model, the execution model, can be described by means of a network of interactive processes, see figure 4.

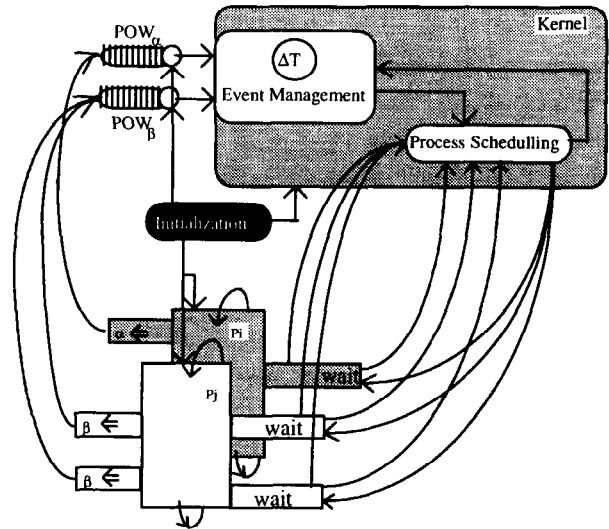


Figure 4. Intermediate Executable Model of VHDL.

In this model, there is one process for each VHDL process resulting from the elaboration, and a new process, named kernel process, which represents the simulator. All these processes are composed by sequential statements, that are cyclically executed. Processes can also contain subprograms.

The communication between these processes is done through shared variables. In this model there are no signals. Each signal coming from the elaboration of a VHDL description is mapped into three global variables (driving value, effective value and current value) that are updated by the kernel process.

The other processes update the value of the variables representing the drivers of the elaborated processes, whose current values are read by the kernel process. Besides of these shared variables used to communicate data between the processes, there are also two shared

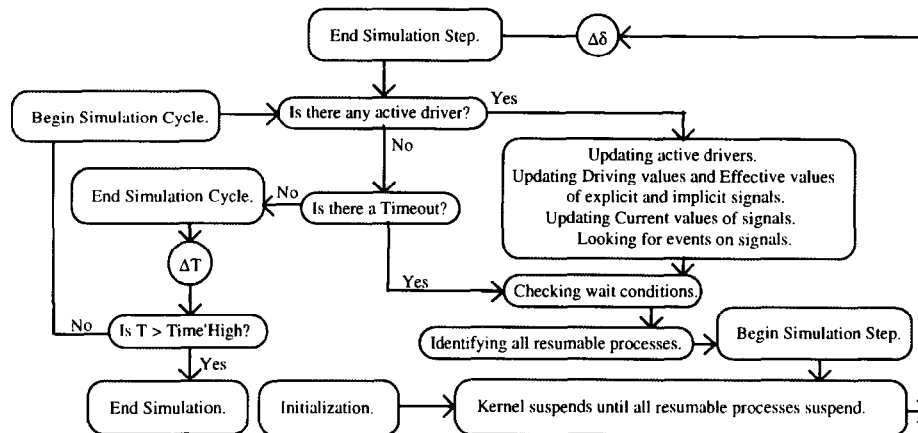


Figure 5.- Control-flow of the kernel process (VHDL simulator).

variables for each elaborated process in order to transfer the execution control between these processes and the kernel process.

The control flow of the kernel process manages the advance of a global variable corresponding with the simulation time, see figure 5. It has two loops, one loop representing the advancement of a delta delay for each simulation step and other loop representing the advancement of the VHDL physical time. This value is used to stop the execution of this model.

The intermediate model is a self-contained model that can be executed by a computer if the processes are expressed in the language of the host computer. The effect of executing any statement of these processes modifies the control flow of the process execution or the value of the shared variables between the processes.

```
entity test is end;
architecture bench of test is
signal x: bit;
signal y: bit_vector(0 to 2);
component cont_3 port(x: in bit; y: out bit_vector(0 to 2));
end component;
for all: cont_3 use entity work.cont_3(structural);
begin
    main : cont_3 port map(x,y);
end;
```

```
entity cont_1 is port(X: in bit; Y: out bit);
end cont_1;
architecture behavioral of cont_1 is
begin
    process
    begin
        wait until x='0';
        y<='1' after 1 ns;
        wait until x='0';
        y<='1' after 1 ns;
    end process;
end behavioral;

entity cont_3 is port (X: in bit; Y: out bit_vector( 0 to 2) );
end;
architecture structural of cont_3 is
    component cont_1 port(X: in bit; Y: out bit);
    end component;
    for all: cont_1 use entity work.cont_1(behavioral);
    signal s: bit_vector(0 to 2);
begin
    y<=s after 1 ns;
    one : cont_1 port map (X,S(0));
    two : cont_1 port map (S(0),S(1));
    three : cont_1 port map (S(1),S(2));
end structural;
```

Figure 6-a.- The example has been included into a VHDL test bench.

It is possible to obtain a formal description of the intermediate software model representing its elements by means of a formal notation, such as Coloured Petri Nets (CPNs), [5]. Taking into account the form in which the intermediate model of the language is executed and the distinguished elements of this intermediate model we present the translation to CPNs in the following steps:

- 1) *Translation of variables, types and expressions.*
- 2) *Translation of the sequential statement of each process.* Previously, the statements corresponding to the algorithms of the kernel process have been expressed in a pseudo-VHDL code in order to use the same translation rules for any process of the intermediate model.
- 3) *Translation of processes.*

First results of this approach were presented in [6].

Figures, 6-a and 6-b, present the processes model corresponding to the VHDL example provided by Carlos Delgado for this book. This example has been modified by Guillermo Ricalde, one of the tool developers presented in section 6, in order to be a simulatable description and be able to produce these two figures.

4. Variables, Types and Expressions.

We assume the reader is familiar with the concepts and notations on CPNs proposed by Kurt Jensen in [5]. For any extension or application of this kind of nets the reader is referred to the previous reference.

4.1. Variables.

We have adopted the following solution for the representation, in CPN terms, of the variables used in the intermediate model of the language:

a) All variables needed in the intermediate model are represented through coloured tokens. These coloured tokens have two components. The first one is an identifier (or tag) that corresponds to the name to the variable. The second one corresponds to the value of the variable and belongs to the type of the variable.

b) There exist an unique place named VHDL_VAR containing all tokens representing variables of the intermediate model of the language. The

colour domain of this place is: $Var_name \times (\bigcup_{\forall var} Type_{var})$

where Var_name is an enumerated type containing all names of variables that can appear in the intermediate model, and $\bigcup_{\forall var} Type_{var}$ is the union of all types of variables of the intermediate model. The type $Index_call$ is defined as **type Index_call is INTEGER range 0 to integer'High;** and $Process_Name$ is an enumerated type containing all CPN_names of processes plus the empty name.

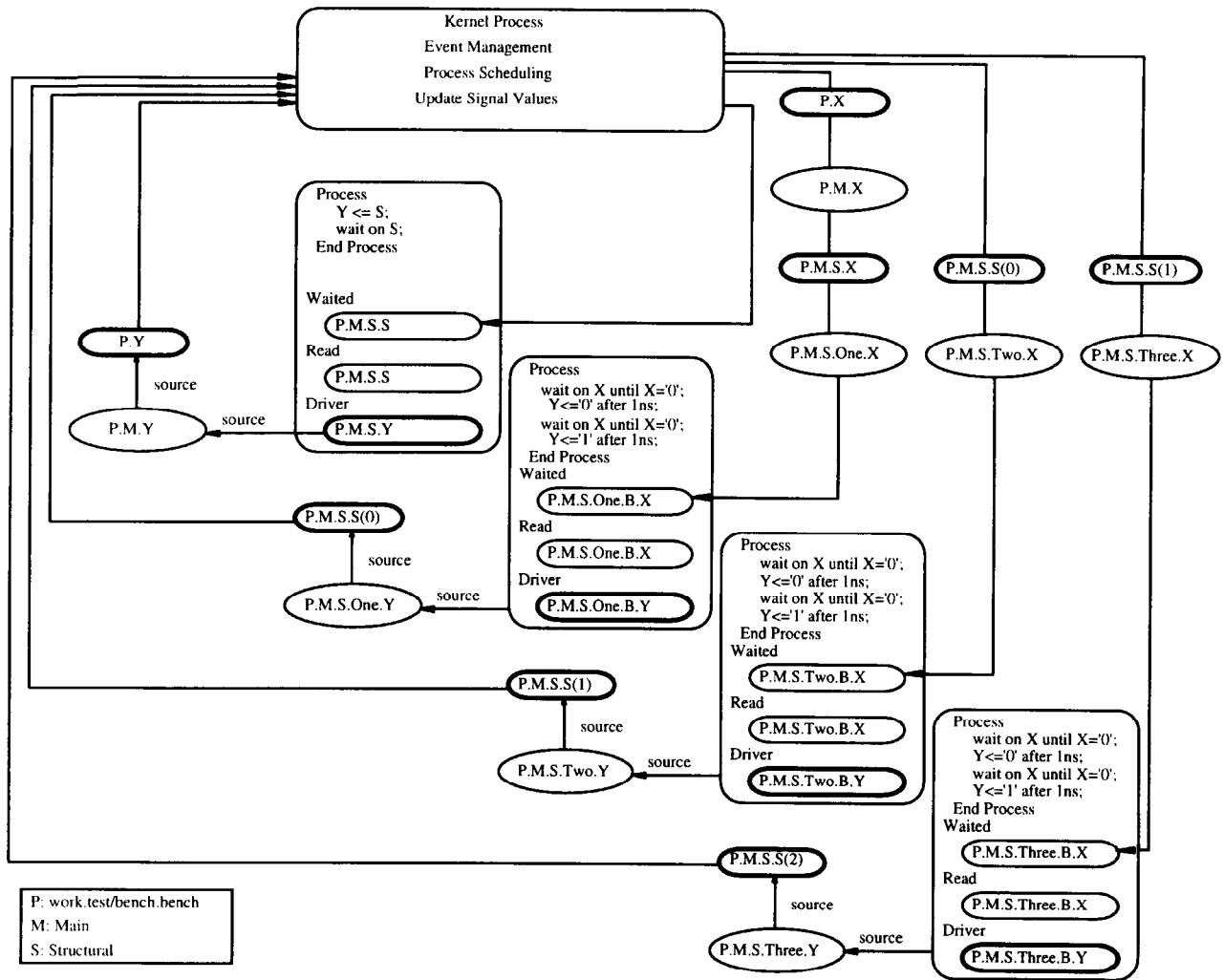


Figure 6-b.- Processes' model corresponding to the example of figure 6-a.

c) A read or write operation on a variable of the intermediate model of a VHDL description is a read or write operation on the second component of a token contained in place VHDL_VAR which first component is the name of variable to be read or written.

d) In order to create a new variable (e.g. the local variables of procedures or functions) a new token will be added to place VHDL_VAR whose first component is the name of the variable and the second component is the initialization value of the variable. The destruction of a variable consist of the removing of the associated token in place VHDL_VAR.

4.2. Generalities on types of the CPN.

The set of types (colour sets) that belong to the set Σ of the coloured Petri Net it is formed by two fundamental subsets: the first one is composed for all elaborated subtypes declared into the VHDL description or implicitly declared in VHDL; the second one is a set of types specifically defined here to consider some special

features of the translation of the intermediate model into CPNs. This second group of types concern transactions, projected output waveforms, index ranges and unconstrained arrays.

A type is characterized by a set of values and a set of operations. The set of operations of a type includes those explicitly declared through subprograms that have a parameter, or result of the type. These kind of operations requires an explicit representation in net terms of the corresponding subprogram. The remaining operations of a type are the predefined operators. These operations are implicitly declared for a given type declaration.

4.2.1. Scalar types

Scalar types consist of enumeration types, integer types, physical types, and floating point types. Enumeration types and integer types are called discrete types. Integer types, floating point types, and physical types are called numeric types. All scalar types are ordered; that is, all relational operators are predefined. Each value of a

discrete or physical type has a position number which is an integer value.

The identifiers and character literals listed by an enumeration type definition must be distinct. Each enumeration literal specification is the declaration of the corresponding enumeration literal. Each enumeration literal yields a different enumeration value. The predefined order relation between enumeration values follow the order of the corresponding position numbers. The position number of the value of the first listed enumeration literal is zero; the position number for each additional enumeration literal is one more than that of its predecessor in the list.

The predefined enumeration types are those from the language: CHARACTER, BIT, BOOLEAN. An additional predefined enumeration type is the type CONTROL: $\text{CONTROL} \in \Sigma$ is $\{\{\bullet\}\}$. This type will be used as base type for uncolored places of a given subnet. Another predefined enumeration type is the type TIMING_MODEL: $\text{TIMING_MODEL} \in \Sigma$ is $\{\text{inertial}, \text{transport}\}$.

An integer type definition defines an integer type whose set of values include those of the specified range. This type class coincides with the corresponding type of the language. The only predefined type is the type INTEGER. The range of INTEGER will be determined after the elaboration of the VHDL description, taking into account the particular implementation constraints of the VHDL environment.

Physical type values represent a measurement of a quantity. Any physical type value is an integral multiple of the base unit of measurement for that type. This type class coincides with the corresponding type of the language. The only predefined physical type is type TIME. The range of TIME will be determined after the elaboration of the VHDL description, taking into account the particular implementation constraints of the VHDL environment. This constant value is Time'High.

Floating point types provide approximations to the real numbers. This type class coincides with the corresponding type of the language. The only predefined floating point type is type REAL. The range of REAL will be determined after the elaboration of the VHDL description, taking into account the particular implementation constraints of the VHDL environment.

4.2.2. Composite types

Composite types are used to define collections of values. These include both arrays of values (collections of values of a homogeneous type) and records of values (collections of values of potentially heterogeneous types).

An array object is a composite object consisting of elements that have the same subtype. The name for an element of an array uses one or more index values belonging to specified discrete types. The value of an array object is a composite value consisting of the values of its elements.

This type class coincides with the constrained_array type of the language. The predefined array types are STRING and BIT_VECTOR. The values of the predefined type STRING are one-dimensional arrays of the predefined type CHARACTER, indexed by values of the predefined subtype POSITIVE (**subtype POSITIVE is INTEGER range 1 to integer_high**);). The values of the predefined type BIT_VECTOR are one-dimensional arrays of the predefined type BIT, indexed by values of the predefined subtype NATURAL (**subtype NATURAL is INTEGER range 0 to integer'High**);).

A record type is a composite type, whose objects consist of named elements. The value of a record object is a composite value consisting of the values of its elements. This type class coincides with the corresponding type of the language.

4.2.3. Transactions, strings of transactions and timing models.

In this subsection we define the basic types of the objects concerning signals and the operations on them. Let σ be a scalar signal with base type $\text{Type}_\sigma \in \Sigma$.

A driver for a scalar signal of a specified type is represented by a *projected output waveform*. A projected output waveform consists of a sequence of one or more *transactions*, where each transaction is a pair consisting of a value component belonging to the base type of the signal and a time component belonging to the predefined type TIME. Transaction and waveform types are useful for support transactions and projected output waveforms concerned in the execution of signal assignment statements.

4.2.3.1. Definition of the type transaction

A transaction object is a composite object consisting of a value component and a time component. The value of a transaction object is a composite value consisting of the values of its elements. For a given transaction, the value component represents a value that the signal driver will assume at the time specified by the time component.

$\text{Transaction}_\sigma = \text{Type}_\sigma \times \text{Time}$.

A signal assignment statement on guarded signals can produce a kind of transactions called *null transactions*. These transactions specify that the driver of the signal is to be turned off, so that it (at least temporarily) stops

contributing to the value of the target. For these signals a transaction object is defined as:

$$\text{Transaction}_\sigma = (\text{Type}_\sigma \cup \{\mathbf{null}\}) \times \text{Time}.$$

where **null** denotes the special value corresponding to null transactions of guarded signals.

We define the following two functions on transactions of σ

$$\text{Value}_\sigma: \text{Transaction}_\sigma \rightarrow \text{Type}_\sigma, \text{ such that } \text{Value}_\sigma((x, t)) = x.$$

$$\text{Time}_\sigma: \text{Transaction}_\sigma \rightarrow \text{Time}, \text{ such that } \text{Time}_\sigma((x, t)) = t.$$

4.2.3.2. Definition of the type string of transactions.

A waveform object is a composite object consisting of a queue of transaction objects. The queue of a waveform object can contain zero transactions, it is said to be empty, and is denoted by the literal **empty**. The value of a waveform object is a composite value consisting of the values of its elements or **empty**. In a non-empty waveform the transactions are ordered with respect to their time components.

The waveform objects are used in the representation of projected output waveforms that they are the representation of the driver of a signal. In the following paragraphs we define the type string of transactions on σ with length less than or equal to k (String_σ^k), where k represents the value of Time'High or the user-defined simulation time.

The type String_σ^k is the following set of elements:

$$\text{String}_\sigma^k = \{\mathbf{empty}\} \cup \{[(x_1, t_1)..(x_n, t_n)] \mid t_n \leq k, (x_j, t_j)$$

$$\in \text{Transaction}_\sigma \forall j \in \{1..n\}, \text{ and } t_j < t_{j+1} \forall i \in \{1..n-1\}\}$$

$$\text{Also we denote, } \text{String}_\sigma^{k*} = \text{String}_\sigma^k \setminus \{\mathbf{empty}\}.$$

Considering the types $\text{Transaction}_\sigma$ and String_σ^k defined above we can construct the type corresponding to a projected output waveform:

$$\text{Projected_waveform}_\sigma = \text{Transaction}_\sigma \times \text{String}_\sigma^k.$$

Therefore, the characteristics of signal drivers are preserved with this representation of a driver:

1) A driver always contains at least one transaction (represented by the first component of values belonging to $\text{Projected_waveform}_\sigma$). The initial contents of a driver associated with a given signal is defined by the default value associated with the signal (default value of the first component). The initial transaction contained in a waveform object is determined according to the rules specified in the language to define the default value associated with a signal.

2) For any driver, there is exactly one transaction whose time component is not greater than the current simulation time (the value of the first component). The current value of the driver is the value component of this transaction. If, as the result of the advance of time, the current time becomes equal to the time component of the first transaction in the second component, then the first component becomes the first transaction of the second component.

The effect of execution of a signal assignment statement is defined in terms of its effect upon the projected output waveforms representing the current and future values of drivers of signals.

The future behavior of the driver(s) for a given target is defined by transactions produced by the evaluation of waveform elements in the waveform of a signal assignment statement.

4.2.3.3. Some functions on string of transactions

Function length (L) on strings of transactions on σ :

$$L: \text{String}_\sigma^k \rightarrow \mathbb{N}, \text{ such that } L(\mathbf{empty}) = 0 \text{ and } L([(x, t)..(x_n, t_n)]) = n.$$

Operation of concatenation of two strings of transactions on σ : $s_1, s_2 \in \text{String}_\sigma^k$.

$$s_1 = [(x_1, t_1)..(x_n, t_n)]; s_2 = [(y_1, r_1)..(y_m, r_m)].$$

$$\oplus: \text{String}_\sigma^k \times \text{String}_\sigma^k \rightarrow \text{String}_\sigma^k$$

such that,

$$1) \mathbf{empty} \oplus s_1 = s_1 \oplus \mathbf{empty} = s_1.$$

$$2) \mathbf{empty} \oplus \mathbf{empty} = \mathbf{empty}.$$

3) $s_1 \oplus s_2 = [(x_1, t_1)..(x_n, t_n)(y_1, r_1)..(y_m, r_m)]$
iff $t_n < r_1$.

Function Head on strings of transactions on σ .

$$H_\sigma: String_\sigma^k \rightarrow String_\sigma^k$$

such that,

1) $H_\sigma([(x_1, t_1)..(x_n, t_n)]) = \text{empty} \in String_\sigma^k$
iff $L([(x_1, t_1)..(x_n, t_n)]) = 0$

2) $H_\sigma([(x_1, t_1)..(x_n, t_n)]) = [(x_1, t_1)] \in String_\sigma^k$
iff $L([(x_1, t_1)..(x_n, t_n)]) > 0$

Function String without Head on strings of transactions on σ .

$$H_\sigma: String_\sigma^k \rightarrow String_\sigma^k$$

such that,

1) $\text{Str}_H_\sigma([(x_1, t_1)..(x_n, t_n)]) = \text{empty} \in String_\sigma^k$
iff $L([(x_1, t_1)..(x_n, t_n)]) \leq 1$

2) $\text{Str}_H_\sigma([(x_1, t_1)..(x_n, t_n)]) = [(x_2, t_2)..(x_n, t_n)]$
 $\in String_\sigma^k$ **iff** $L([(x_1, t_1)..(x_n, t_n)]) > 1$

Function Bottom on strings of transactions on σ .

$$B_\sigma: String_\sigma^k \rightarrow String_\sigma^k$$

such that,

1) $B_\sigma([(x_1, t_1)..(x_n, t_n)]) = \text{empty} \in String_\sigma^k$ **iff**
 $L([(x_1, t_1)..(x_n, t_n)]) = 0$

2) $B_\sigma([(x_1, t_1)..(x_n, t_n)]) = [(x_n, t_n)] \in String_\sigma^k$
iff $L([(x_1, t_1)..(x_n, t_n)]) > 0$

Function String without Bottom on strings of transactions on σ .

$$\text{Str}_B_\sigma: String_\sigma^k \rightarrow String_\sigma^k$$

such that,

1) $\text{Str}_B_\sigma([(x_1, t_1)..(x_n, t_n)]) = \text{empty} \in String_\sigma^k$ **iff** $L([(x_1, t_1)..(x_n, t_n)]) \leq 1$

2) $\text{Str}_B_\sigma([(x_1, t_1)..(x_n, t_n)]) = [(x_1, t_1)..(x_{n-1}, t_{n-1})] \in String_\sigma^k$ **iff** $L([(x_1, t_1)..(x_n, t_n)]) > 1$

Function Transaction Bottom on non-empty strings of transactions on σ .

$$\text{TB}_\sigma: String_\sigma^{k*} \rightarrow \text{Transaction}_\sigma$$

such that,

$\text{TB}_\sigma([(x_1, t_1)..(x_n, t_n)]) = (x_n, t_n) \in \text{Transaction}_\sigma$

Temporal increment of the time part of the transactions in a string of transactions on σ (this operation contains also the truncation of the string if the result of the increment exceeds the Time'High or the user-defined simulation time k).

$$\otimes: \text{Time} \times String_\sigma^{k*} \rightarrow String_\sigma^{k*}$$

such that, given $l \in \text{Time}$ and $s = [(x_1, t_1)..(x_n, t_n)] \in String_\sigma^{k*}$.

1) $l \otimes s = [(x_1, t_1 + l)..(x_n, t_n + l)]$ **iff** $t_n + l \leq k$.

2) $l \otimes s = [(x_1, t_1 + l)..(x_i, t_i + l)]$ **iff** $i < n$, $t_i + l \leq k$ and $t_{i+1} + l > k$.

4.2.3.4. Functions implementing the timing models of VHDL.

Time Filter function of a string of transactions on σ w.r.t. t (TIME).

$$\text{TF}_\sigma^t: String_\sigma^k \rightarrow String_\sigma^k$$

such that, given $s = [(x_1, t_1)..(x_n, t_n)] \in String_\sigma^{k*}$, and $t \in \text{TIME}$.

1) $\text{TF}_\sigma^t(\text{empty}) = \text{empty}$.

2) $\text{TF}_\sigma^t(s) = (\text{empty})$ **iff** $\text{Time}_\sigma(\text{TH}_\sigma(s)) \geq t$.

3) $\text{TF}_\sigma^t(s) = H_\sigma(s) \oplus \text{TF}_\sigma^t(\text{Str}_H_\sigma(s))$ **iff** $\text{Time}_\sigma(\text{TH}_\sigma(s)) < t$.

Value Filter function of a string of transactions on σ w.r.t. v .

$$VF_{\sigma}^t: String_{\sigma}^k \rightarrow String_{\sigma}^k$$

such that, given $s = [(x_1, t_1)..(x_n, t_n)] \in String_{\sigma}^{k*}$, and $v \in Type_{\sigma}$.

$$1) VF_{\sigma}^t(\text{empty}) = \text{empty}.$$

$$2) VF_{\sigma}^t(s) = (\text{empty}) \text{ iff } Value_{\sigma}(TB_{\sigma}(s)) \neq v.$$

$$3) VF_{\sigma}^t(s) = VF_{\sigma}^t(\text{Str_B}_{\sigma}(s)) \oplus VF_{\sigma}^t(\text{B}_{\sigma}(s)) \text{ iff } Value_{\sigma}(TB_{\sigma}(s)) = v.$$

Transport composition operation of two strings of transactions on σ , s' and $s'' \in String_{\sigma}^k$.

This is a binary operation that implements the composition of two waveforms under the transport timing model (in a non-empty waveform the transactions are ordered with respect to their time components). The composition consists of the deletion of zero or more transactions of the first operand (called old transactions), and the addition of the transactions of the second operand, as follows:

(1) All old transactions whose time components are greater than or equal to the time component of the first transaction belonging to the second operand are deleted from the first operand;

(2) The transactions of the second operand are then appended to the transactions of the first operand.

$$\oplus_t^{\sigma}: String_{\sigma}^k \times String_{\sigma}^k \rightarrow String_{\sigma}^k$$

such that,

$$s' \oplus_t^{\sigma} s'' = TF_{\sigma}^t(s') \oplus s''$$

where $t = \text{Time}_{\sigma}(\text{TH}_{\sigma}(s''))$.

Inertial composition operation of two strings of transactions on σ , s' and $s'' \in String_{\sigma}^k$.

This is a binary operation that implements the composition of two waveforms under the inertial timing model (in a non-empty waveform the transactions are ordered with respect to their time components). The resulting waveform from this kind of composition is obtained as follows:

(1) All of the transactions belonging to the second operand are marked. They are then appended to the transactions of the first operand (called old transactions) that remain unmarked;

(2) An old transaction is marked if it immediately precedes a marked transaction and its value component is the same as that of the marked transaction;

(3) All unmarked transactions (all of which are old transactions) are deleted from the result waveform.

$$\oplus_i^{\sigma}: String_{\sigma}^k \times String_{\sigma}^k \rightarrow String_{\sigma}^k$$

such that,

$$s' \oplus_i^{\sigma} s'' = VF_{\sigma}^t(\text{F}_{Pr2}^{\sigma}(s')) \oplus s''$$

where

$$v = Value_{\sigma}(\text{TH}_{\sigma}(s'')),$$

$$t = \text{Time}_{\sigma}(\text{TH}_{\sigma}(s'')) \text{ and}$$

4.2.3.5. Some functions on projected waveforms

Function *TIME_HEAD* on Projected Waveforms on σ :

$$TIME_HEAD: \bigcup_{\sigma \in \text{Scalar_Signal_Name}} Projected_waveform_{\sigma} \rightarrow Time$$

such that,

$$1) TIME_HEAD(((x,y), \text{empty})) = \text{Time'High}$$

$$2) TIME_HEAD(((x,y), [(x_1, t_1)..(x_n, t_n)])) = t_1$$

Function *VALUE_HEAD* $_{\sigma}$ on Projected Waveforms on σ :

$$VALUE_HEAD_{\sigma}: Projected_waveform_{\sigma} \rightarrow Type_{\sigma}$$

such that, $VALUE_HEAD_{\sigma}(((x,y), [(x_1, t_1)..(x_n, t_n)])) = x$

Function *Update_Driver_CV_Type* $_{\sigma}$ on Projected Waveforms on σ :

$$Update_Driver_CV_Type_{\sigma}: Projected_Waveform_{\sigma} \rightarrow Projected_Waveform_{\sigma}$$

such that,

1) $\text{Update_Driver_CV_Type}_\sigma(((x,y),\text{empty})) = ((x,y),\text{empty})$.

2) $\text{Update_Driver_CV_Type}_\sigma(((x,y), [(x_1, t_1)..(x_n, t_n)])) = ((x_1, t_1), [(x_2, t_2)..(x_n, t_n)])$.

4.2.4. Unconstrained array types and Index ranges.

In order to consider the different discrete ranges on which a loop parameter can take values we need a new type associated to the place that will contain this dynamically elaborated discrete range.

A index range type defined over the range base type T is defined as

$I_T: T^2 \times \text{Sense}$

where $\text{Sense} = \{\text{to}, \text{downto}\}$.

A index range I_T defines a subset of the set of all values of the type T as follows. Let $I_T = (a, b, s)$ be an index range where $a, b \in T$ and $s \in \text{Sense}$. Since T is a scalar type let \leq_T be the associated total order relation. Then, defines the set of values of type T:

1. If $a \leq_T b$ and $s = \text{to}$, then $I_T = \{ v \in T \mid a \leq_T v \leq_T b \}$
2. If $b \leq_T a$ and $s = \text{downto}$, then $I_T = \{ v \in T \mid b \leq_T v \leq_T a \}$
3. $I_T = \emptyset$ in other cases

Other kind of objects that we must considered in the translation of the intermediate model of the language into coloured Petri nets is the unconstrained arrays. Therefore, we define a new type for considering these objects.

An array is characterized by the number of indices (the dimensionality of the array), the type, position, and range of each index, and the type and possible constraints of the elements. The order of the indices is significant.

An unconstrained array definition defines an array and a name denoting that type. For each object that has the array type, the number of indices, the type and position of each index, and the subtype of the elements are as in the type definition. The index subtype for a given index position is, by definition, the subtype denoted by the type mark of the corresponding index subtype definition. The values of the left and right bounds of each index

range are not defined but must belong to the corresponding index subtype; similarly, the direction of each index range is not defined.

A constrained array definition defines both an unconstrained array type and a subtype of this type:

1. The array type is an implicitly declared anonymous type; this type is defined by an (implicit) unconstrained array definition, in which the element subtype indication is that of the constrained array definition, and in which the type mark of each index subtype definition denotes the subtype defined by the corresponding discrete range.

2. The array subtype is the subtype obtained by imposition of the index constraint on the array type.

According to the above rules now we define the unconstrained array type. Let $U(I_{T_1}, \dots, I_{T_k}, D)$ be an unconstrained array type whose indices are of the index range types I_{T_1}, \dots, I_{T_k} and so that each element of the array is of type D. Given such a unconstrained type, we will assume the existence of:

1. A constant value named *null array*, $\text{null}_{I_{T_1}, \dots, I_{T_k}, D}$, as defined in the LRM (any of the discrete ranges of the array defines a null range).
2. A function Ψ_U defined over the set of all constrained arrays of the form

$C((I_{T_1}, R_1), \dots, (I_{T_k}, R_k), D)$

whose indices are of the index range types I_{T_1}, \dots, I_{T_k} and R_1, \dots, R_k define the index constraints, and giving values in $U(I_{T_1}, \dots, I_{T_k}, D)$ so that: $\Psi_U(C) = C_U$ verifying

a) $\forall \text{index} \in \{1, \dots, k\}, \text{Range}(C_U, \text{index}) = \text{Range}(C, \text{index})$

b) $\forall \text{index} \in \{1, \dots, k\}, \forall r_i \in R_i, C_U[r_1, \dots, r_k] = C[r_1, \dots, r_k]$

c) $\forall \text{index} \in \{1, \dots, k\}, \forall r_i \in T_i \setminus R_i, C_U[r_1, \dots, r_k]$ is undefined

4.3. Function-free Expressions.

A VHDL expression is a formula that defines the computation of a value. The kind of expressions and operands, their syntax, and their semantics are very close to those stated in the elaborated model. In general, the translation of an elaborated expression into CPN leads to an arc expression labeling the arc from the transition representing the evaluation of the expression and the

place VHDL_VAR. The effect of this arc expression is to insert a token into the place VHDL_VAR where the first component represents the *CPN_name* of the variable containing the value of the expression and the second component represents the value itself. Some exceptions exist to this rule, e.g. when function call appear as part of an expression. Because the above general principle for the translation of an elaborated expression, the translation of expressions into CPNs mainly concerns the definition of the syntax rules to describe arc expressions and their semantics. These rules are very close to those defined in the elaborated language (in essence, expressions and their corresponding arc expressions realize the same computations).

In order to translate an elaborated expression into a Coloured Petri Net two steps are needed.

The first step generate an intermediate code we call function-free code. Function-free code is a sequence of statements of the general form: $x := F_expression$; where *F_expression* is an elaborated expression without calls to user-defined functions (it can appear predefined functions whose parameters are names) or one call to a user-defined function which parameters are names; and *x* is a name, or translator-generated temporaries. Thus, a source language expression like

$$(2 * x + \text{Funcion1}(\text{Function2}(2 * a, b + c), b)) * (y+z)$$

might be translated into a sequence: $t_1 := \text{Function2}(2 * a, b + c)$; $t_2 := \text{Funcion1}(t_1, b)$; $value := (2 * x + t_2) * (y+z)$.

where t_1 , t_2 and *value* are translator-generated temporary names. This unraveling of complicated VHDL expressions (and later of nested flow-of-control statements) makes function-free code desirable for CPN generation and optimization. The use of names for intermediate values computed by any expression allows function-free code to be easily rearranged (if needed).

Function-free code is a linearized representation of a compacted version of a syntax tree of a dag in which *F_expressions* and names correspond to the interior nodes of the graph.

Statements can have symbolic labels and there are statements for flow control. Here are the common function-free statements used in the remainder of this document (only 1,2 and 6 are needed in this section):

1. Assignment statements of the form $x := F_expression$.

2. Copy statements of the form $x := y$ where the value of *y* is assigned to *x*.
3. The unconditional jump *goto L*.
4. Conditional jumps such as *if F_boolean_expression goto L*.
5. Procedure call, *call procedure(F_expression₁, ..., F_expression_n)*.
6. Indexed assignments of the form $x := y[i]$ and $x[i] := y$.

When function-free code is generated, temporary names are made up for the interior nodes of a syntax tree.

The second step for the translation of elaborated expressions consist of the generation of the Coloured Petri Nets corresponding to the sequence of assignments (belonging to function-free code) in which the original expression has been decomposed. The rules for the translation are:

1. For each assignment statement in which the right-hand side is an elaborated expression without calls to user-defined functions:

a) Generate a transition with suffix name equal to *.Simple_Assignment*, an input place with suffix name *.Begin_Assignment* and output place with suffix name *.End_Assignment*. The colour domain of these places is the type CONTROL.

b) Add two arcs: the first one, for the place VHDL_VAR to transition *.Simple_Assignment*, reads the token representing the variable; the second one, from the transition *.Simple_Assignment* to the place VHDL_VAR, inserts a token (by means of the corresponding arc expression) which first component is the *CPN_name* of the variable and the second one is the expression of the right-hand side of the assignment being translated.

c) Add to the arc expressions of the above two arcs the read operation of all variables appearing in the right-hand expression of the assignment

The above rules assume that in the definition of the colour domain of the place VHDL_VAR has been taken into account the *CPN_names* and types of the temporary variables. The permanent existence or dynamic creation of these temporary variables depends on the scope of their declaration.

2. For each assignment statement in which the right-hand side is a call to a user-defined function whose

parameters do not include calls to user-defined functions, generate a call subnet as defined in sections 5 and 7. The transition `.RETURN` of this subnet inserts the returned value of the function in the variable appearing in the left-hand side of the assignment.

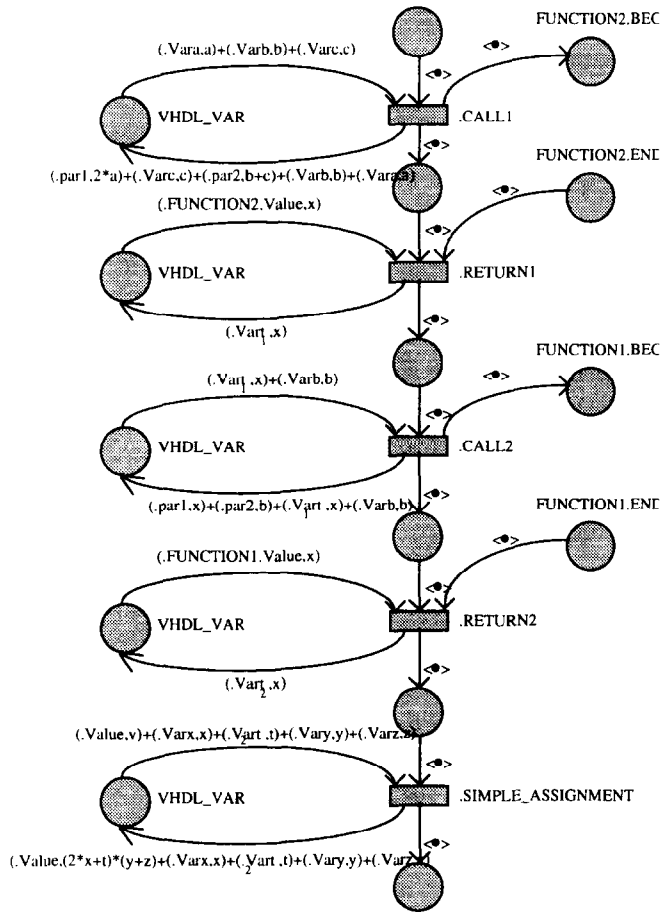


Figure 7.- CPN representation of a VHDL function-free expression.

3. In order to compose the full net for the elaborated expression the subnets generated for the assignments in which the expression has been decomposed must be put together. To do this, make the fusion of a place `.End_Assignment` with the `.Begin_Assignment` place or `.Begin_call` place of the subnet of the following assignment. The name of the fused place will be `.Intermediate_Assignment` (with colour domain of type CONTROL). The figure 7 presents the subnet generated for the example presented above.

5. Statements, Subprograms and Processes.

5.1. Statements.

The result of the translation of sequential statements into CPNs leads to subnets where we can distinguish two fundamental parts:

- The control flow corresponding to the considered sequential statement.
- The different inscriptions of the subnet representing the operations carried out on variables, signals and so on.

The first part give rise to safe subnets, with at most one token belonging to the type CONTROL, that we call control flow skeletons of the sequential statements. These control flow skeletons have only one input place (representing the begin of the statement) and only one output place (representing the end of the statement). The different firing sequences moving the token from the input place to the output place, represent the different ways of executing a given sequential statement. The second part of the description of sequential statements in CPN terms concerns operations carried out on variables or signals, parameter passing in procedure call, and so on. For the translation of these parts we must consider the rules developed in precedent sections for the translation of elaborated expressions.

In figure 8 we present examples of the results obtained in the translation of sequential statements different to the signal assignment statement and the wait statement. These two statements are considered in the following subsections because their importance in the communication between user-defined processes and kernel process.

5.1.1. Assignment Statement.

The only difference between a signal assignment statement and a variable statement (see figure 7) consist on evaluating the projected waveform corresponding to the right-hand side of the assignment. Evaluation of a waveform consists of the evaluation of each waveform element in the waveform. Thus the evaluation of a waveform results in a sequence of transactions, where its transaction corresponds to one element in the waveform. These transactions are called new transactions. The resulting waveform must has its elements in ascending order with respect to time, otherwise it is an execution error that is shown in the CPN, see figure 9.

5.1.2. Wait Statement.

The execution of a wait statement causes the time expression to be evaluated to determine the *timeout interval*. It also causes the execution of the corresponding process to be suspended, where the corresponding process statement is the one that either contains the wait statement or is the parent of the procedure that contains the wait statement. The suspended process will resume, at the latest, immediately after the timeout interval has expired.

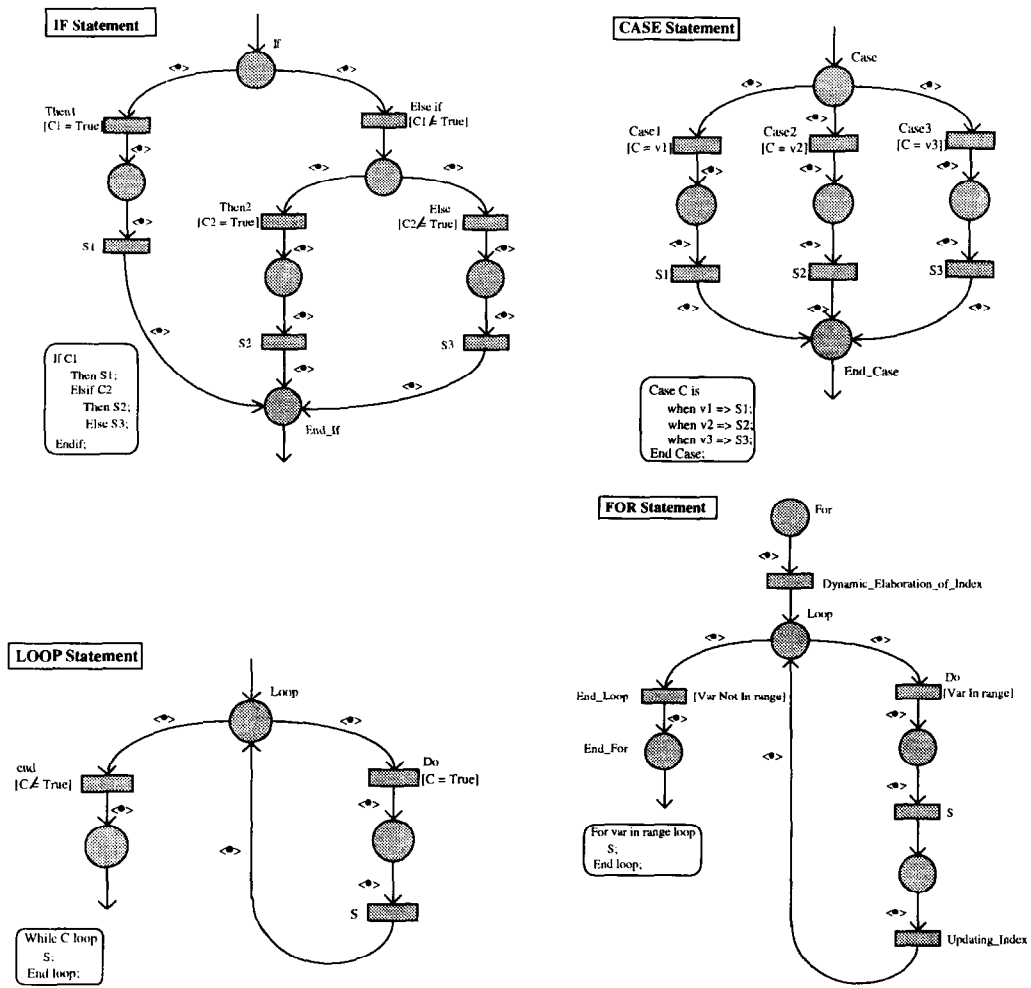


Figure 8.- Main control statements.

The suspended process may also resume as a result of an event occurring on any signal in the sensitivity set of the wait statement. If such an event occurs, the condition in the condition clause is evaluated. If the value of the condition is TRUE, the process will resume. If the value of the condition is FALSE, the process will re-suspend. Such re-suspension does not involve the recalculation of the timeout interval.

Figure 10 presents the CPN representation of a wait statement. The process containing the wait statement communicate to the kernel process the list of signals to which it is sensitive when suspends its execution flow marking the .Suspend place. When the kernel process detect an event on some of these signals then the control is passed to the subnet corresponding to the wait statement in order to compute the wait condition. The boolean result is transferred to the kernel process in order that this process can identify the resumable processes for next simulation cycle. These resumable processes includes those complying their timeout conditions. The subnet corresponding to the kernel process returns the execution flow to the other processes by means of the .Resume place when a process is resumable.

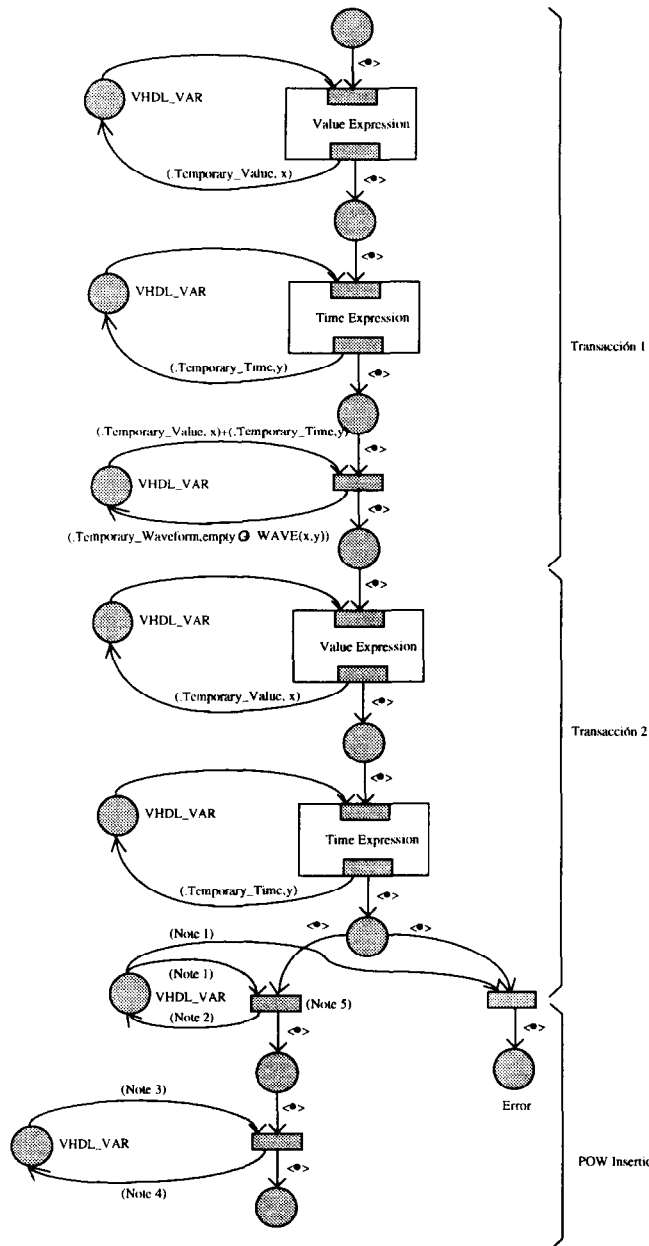
The CPN of a wait statement represents the control flow communication between processes of the intermediate model.

5.2. Subprograms.

There are two forms of subprograms: procedures and functions. A procedure call is a statement; a function call is an expression and returns a value. The definition of a subprogram can be given in two parts: a subprogram interface defining its communication conventions with its parent, and a subprogram body defining its execution.

The subprogram body is composed by sequential statements and follows the same rules as presented for processes in next subsection.

The subprogram interface allows parameters and control passing from the parent to the subprogram and returning results and control from the subprogram to its parent. The subprogram call creates temporal variables needed for the subprogram execution that are destroyed when the control flow returns to the subprogram parent.

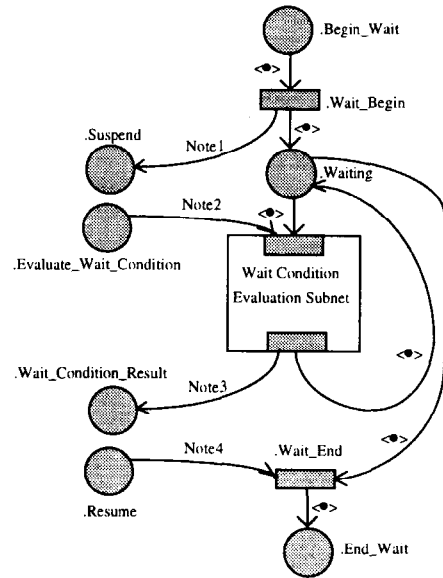


Note1: $(.Temporary_Waveform,x)+(.Temporary_Value,y)+(.Temporary_Time,z)$.
 Note2: $(.Temporary_Waveform,x \oplus WAVE(y,z))$.
 Note3: $(.POW_{\sigma}(x,y))+(.Temporary_Waveform,z)$.
 Note4: $(.POW_{\sigma}(x, Temporary_Type_{\sigma}(y,z)))$.
 Note5: $(TIME_{\sigma}(x, Transaction_Tail(x)) < z)$.
 Note6: $(TIME_{\sigma}(x, Transaction_Tail(x)) \geq z)$.

Figure 9.- CPN representation of a signal assignment statement with two waveform elements.

A subprogram call can be recursive and even reentrant in the case of procedures used concurrently by two parents which parent processes are different.

This complicates the parameter passing to subprograms. Figure 11 presents the CPN corresponding to a simple function described below:



Note 1: $(Process_Name, (Value_For_Expr, \Psi_u(A(Sensitivity_List), False, False))$.
 Note 2: $(Process_Name, (x,y,z,k))$.
 Note 3: $(Process_Name, (x,y, Boolean_Condition_Value, k))$.
 Note 4: $(Process_Name, (x,y,z, True))$.

Figure 10.- CPN representation of a wait statement.

```

FUNCTION MIN(A;B:INTEGER)
  RETURN INTEGER IS
BEGIN
  IF A<B THEN
    RETURN A;
  ELSE
    RETURN B;
  END IF;
END MIN;

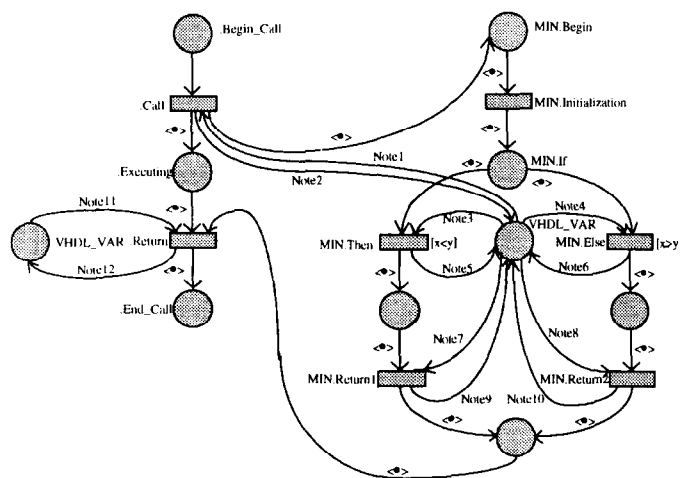
```

5.3. Processes.

In the description of a given process in CPNs we must take into account that we have a piece of sequential code with its own sequential control flow.

Therefore, the global structure of an elaborated process is a Petri net cycle as the example in figure 12. In this figure we represent:

- 1) Each sequential statement (S1, S2, S3, S4) contained in the process P_E is represented by an input place (precondition), a transition representing the computation carried out and an output place representing the postcondition.



- Note1: (.Var1,z).
- Note2: (.Var1,z)+(.MIN.A,z)+(.MIN.B,5).
- Note3: (.MIN.A,x)+(.MIN.B,y).
- Note4: (.MIN.A,x)+(.MIN.B,y).
- Note5: (.MIN.A,x)+(.MIN.B,y).
- Note6: (.MIN.A,x)+(.MIN.B,y).
- Note7: (.MIN.A,x)+(.MIN.B,y).
- Note8: (.MIN.A,x)+(.MIN.B,y).
- Note9: (.MIN.Value,x).
- Note10: (.MIN.Value,y).
- Note11: (.MIN.Value,z)+(.Var1,y).
- Note12: (.Res,z).

Figure 11.- CPNs representation of MIN function.

2) The sequential execution of statements S1, S2, S3 and S4 is represented by means of the fusion of the postcondition of a sequential statement with the precondition of the following sequential statement (in net terms, by the fusion of the places with the same name).

3) The repetitive nature of a process statement is represented by means of the fusion of the postcondition of the last sequential statement in the process with the precondition of the first sequential statement (in net terms, making a cyclic net).

4) The fact that in the initialization phase of the execution of a VHDL description, all processes are actives and execute statements until they reach a wait statement is represented by means of the initial marking in the net. That is, the precondition of the first statement is marked indicating that this is the first sequential statement to be executed. The execution continues by moving the token from place to place until a wait statement is reached.

Therefore, the translation of processes into CPNs must consider two basic parts:

1) The translation of the elaborated declarations of the process. This part creates the needed objects for the process execution.

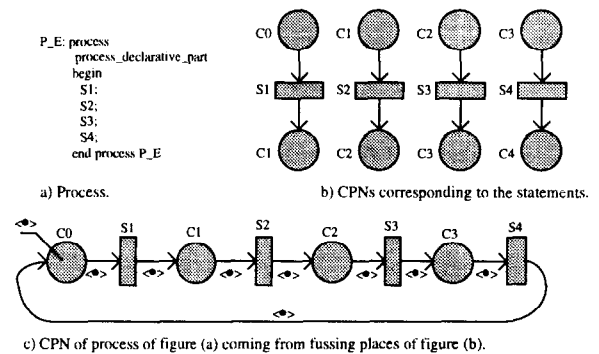


Figure 12.- CPN representation of a process.

2) The translation of the sequential statements of the process. The scheme to implement the translation for the flow-of-control constructs is a syntax-directed translation scheme. Nevertheless, in this document we present the result of the translation for each sequential statement in order to clarify all features of the elaborated model that must be taken into account in the translation.

The kernel process can be expressed in a VHDL-like notation in order to use the same CPN representation rules used with the other processes. This VHDL-like code does not need to be elaborated. Figure 12 presents the CPN corresponding to this process which algorithm was presented in figure 5. This process calls VHDL-like procedures such as: Update_Active_Drivers, Update_Active_Explicit_Signals, Update_Driving_Value s, Update_Effective_Values, Update_Current_Values, Update_Implicit_Signals, Evaluate_Wait_Conditions, Compute_Event_Resumable_Processes, Resume_Processes, Wait_All_Suspended, Compute_Tim e_Resumable_Processes; and VHDL-like functions such as: There_exists_a_Resumable_Process, Min_Active_POW, Min_Timeout.

The notes used in the figure 13 are:

- Note1, Note2, Note3, Note4: (Current_Time,x)
- Note5: (Active_POW_Time,x) + (Min_Active_POW.Value,y)
- Note6: (Active_POW_Time,y)
- Note7: (Timeout_Time,x) + (Min_Timeout.Value,y)
- Note8: (Timeout_Time,y)
- Note9, Note10, Note11, Note12: (Active_POW_Time,x) + (Current_Time,y)
- Note13, Note14, Note15, Note16: (Timeout_Time,x) + (Current_Time,y)
- Note17, Note19: (There_Exists_A_Resumable_Process.Value,x)

Note21, Note22, Note23, Note24:
 (Active_POW_Time,x)+(Timeout_Time,y)
 Note25, Note26, Note27, Note28:
 (Current_Time,x)+(Timeout_Time,y)
 Note29: $[x \geq \text{TIME}'\text{HIGH}]$
 Note30: $[x < \text{TIME}'\text{HIGH}]$
 Note31, Note33: $[x = y]$
 Note32, Note34: $[x \neq y]$
 Note35: $[x]$
 Note36: $[\text{NOT}(x)]$
 Note37: $[x > y]$
 Note38: $[x \leq y]$

6. Implementation of a CPN model Generator.

A set of tools based in a formal model of VHDL [6] is under development in the FORMAT Project (CEC ESPRIT III project #6128). These tools conform a framework for automated static analysis of VHDL [7-9] based on Coloured Petri Nets (CPN). TGI is developing a Petri Net Generator and a Petri Net Analyzer as well as a translator of Petri Nets (PN) into Transition Systems, which is the path to verification tools.

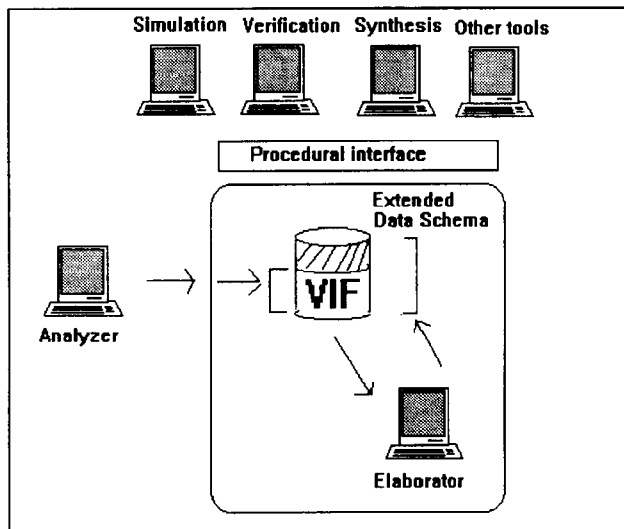


Figure 14.- Extension of commercial VHDL-Front-end.

Any VHDL framework must contain at least an analyzer, an elaborator and a simulator. Tools can be designed from the output of each of them. In the approach presented in this work the starting point for the verification tools is not the output of the analyzer but the output of the elaborator (figure 14) because the VHDL analyzer does not create any executable model. This model is generated by the VHDL elaborator, being in this way an intermediate step for either simulation or verification tools. To enable full concentration on the development of the Petri Net based tools TGI chose to utilize an off-the-shelf front-end analyzer and code generator. The chosen front-end was provided by LEDA S.A, see figure 14.

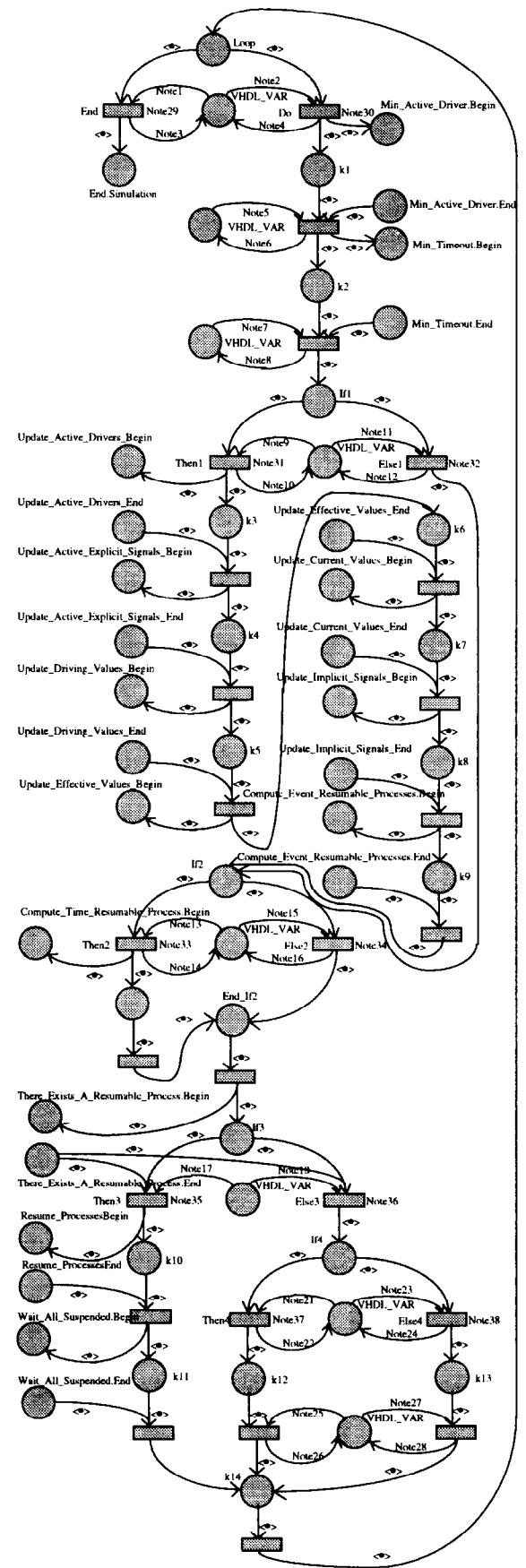


Figure 13.- CPN corresponding to the kernel process.

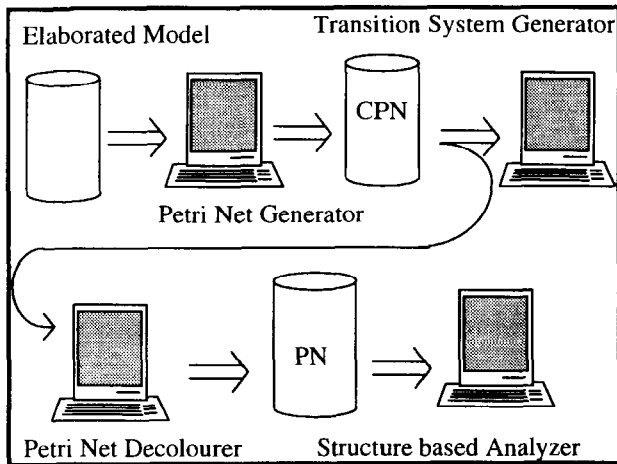


Figure 15.- Petri Net based tools in FORMAT project. Figure 15 shows the architecture of the formal verification tools developed in FORMAT project.

The VHDL elaborator was finished by October, 1993. The Petri Net Generator, finished by June, 1994, produces the CPN representation corresponding to any VHDL description. The other tools are under developing and should be finished by the end of this year.

7. Conclusions.

I have sketched a CPN model of the VHDL execution model. The formal model presented in this paper corresponds to the execution of the elaborated processes by a VHDL simulator. Data, control and time aspects are considered in this model. The relation between this formal model and the syntax can be obtained following the backtracking information associated with the elaboration of a VHDL design stored in a design library resulting from the analysis of the corresponding VHDL design files. To generate the CPN of a VHDL description we start from the elaborated processes. The model also includes a kernel process which represents the simulator. The CPN model is independent of the description style and abstraction level chosen by the designer. This approach can be applied to other HDLs with the same underlying model. From a more general perspective, formal language definition should include also syntax checking and analysis definition, and a formal definition of the rules to elaborate a VHDL source code.

Formal semantics of the language is valuable to CAD tool implementors and designers, for it provides: (1) A precise standard for an implementation. (2) Useful user documentation. (3) A tool for design and analysis, [6-9]. (4) Input to a compiler generator. Further steps of this work will be addressed to complete the semantics' definition of all features supported by the language. Several tools are being developed in FORMAT project.

These tools are: (1) CPN Generator from a VHDL description; (2) State Space Generator of the CPN; (3) Analyzers based on the reachability graph and the structure of the Petri Nets. These tools will help to apply formal verification techniques to VHDL-based hardware designs.

References.

- [1] "IEEE Standard VHDL Language Reference Manual", IEEE, Inc., New York, N.Y., U.S.A., March 1988.
- [2] "IEEE Standards Interpretations: IEEE Std 1076-1987, IEEE VHDL Language Reference Manual", IEEE, Inc., New York, N.Y., U.S.A., 1992.
- [3] "IEEE Standard VHDL Language Reference Manual", IEEE, Inc., New York, N.Y., U.S.A., June 1994.
- [4] S. Olcoz, J. M. Colom, "The Discrete Event Simulation Semantics of VHDL." Intl. Conf. on Simulation and HDLs, pp. 128-134, January 1994, Tempe (Arizona), U.S.A.
- [5] K. Jensen, "Coloured Petri Nets: A High Level Language for System Design and Analysis". In Advances in Petri Nets 1990. Lecture Notes in Computer Science, vol. 483, Springer, Berlin. Heidelberg New York 1990, pp. 342-416.
- [6] S. Olcoz, J. M. Colom, "Toward a Formal Semantics of IEEE Std. VHDL-1076." EuroVHDL'93, Hamburg, 1993, pp. 526-531.
- [7] S. Olcoz, J. M. Colom, "Petri Net Based Analysis of VHDL Programs." 2nd International Conference EuroVHDL-91. Stockholm, September 1991.
- [8] S. Olcoz, J. M. Colom, "A Petri Net Approach for the Analysis of VHDL Descriptions." CHARME'93, Arles, May 1993, pp. 15-26
- [9] S. Olcoz, J. M. Colom, "Analysis Tools applied to VHDL." EUROMICRO'93, Barcelona, September 1993, pp. 597-604.
- [10] D. Borrione, H. Eveking and L. Pierre, "Formal proofs from HDL descriptions." Technical Report 1993.
- [11] P. T. Breuer, L. Sánchez and C. Delgado, "Clean Formal Semantics for VHDL." EDAC 1994.
- [12] W. Damm, B. Josko and R. Schlör, "Linking VHDL with Formal Verification Tools: How to Generate Finite-State Models out of VHDL Designs." Technical Report, University of Oldenburg 1993.
- [13] J. van Tassel, "The Semantics of VHDL with VAL and HOL: Towards Practical Verification Tools." M.Sc. thesis, Dept. of Computer Science and Engineering, Wright University, 1989, also: Technical Report 196, University of Cambridge, Computer Laboratory, UK, June 1990.