VHDL & Signal: A Cooperative Approach

Mohammed Belhadj

▶ **To cite this version:**

Mohammed Belhadj. VHDL & Signal: A Cooperative Approach. International Conference on Simulation and Hardware Description Languages (ICSHDL), Jan 1994, Tempe, Arizona, United States. pp.76-81. hal-00545943

HAL Id: hal-00545943

https://hal.science/hal-00545943

Submitted on 13 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

VHDL & SIGNAL: A COOPERATIVE APPROACH

Mohammed BELHADJ
IRISA
Campus de Beaulieu
35042 RENNES, FRANCE

ABSTRACT

This work presents the study done to integrate VHDL with the SIGNAL environment for reactive systems, signal processing and real-time applications. This will be done by translating the language SIGNAL into VHDL and conversly, VHDL into SIGNAL. As the scope and the use of these two languages are different, a subset of both languages will be defined to correctly interface them. By allowing description in a mixture of VHDL and SIGNAL, designers gain from the strengths of both languages and their tool environments.

1 INTRODUCTION

Rapidly growing complexity of digital systems justifies extensive research activity on the topics of design methodologies. Hardware description languages (HDLs) have made possible productivity gains in the design of VLSI circuits. However, the gap between the possibilities offered by HDL tools (as part of CAE tools) and the capabilities of VLSI is still very large, and there is a need for new design methodologies.

In the past, the design activities like simulation, design entry, etc, was done at the silicon level. The first abstraction step was the move to gate level, then to the RTL level.

Now, good CAE tools for silicon compilation from RTL description exist, generally described by VHDL (LRM 1987) or Verilog code. However, to address today's technology and market constraints (particularly small time to market) we need to move to more abstract tools (i.e at behavioral and system level). This will improve the productivity of circuit design.

The need for new methodologies and tools for high level design of systems motivates the following work. It uses the SIGNAL language (Le Guernic 1991) for system and behavior level design and VHDL for connection with CAD/CAE tools. In the following we present the SIGNAL language and its environment. It is an abstract language for reactive, signal processing and real-time applications. Next, we discuss the choice of interfacing SIGNAL and VHDL. Then, the way we link SIGNAL to VHDL is presented.

2 FEATURES OF SIGNAL

2.1 Signals, Clocks And Events

The basic object handled by SIGNAL is a possibly infinite sequence of values called a *signal*. The set of instants where those values occur is called a *clock*. No universal (absolute) time exists, the set of occurrence instants of a signal is defined relatively to other observed signals. Consider the following (isolated) signal "-1,1,-4,3,2,..." observed together with another signal:

-1	$\perp$	1	$\perp$	-4	3	2	...
$\perp$	0	4	-5	2	$\perp$	1	...

The first signal is absent at instants 2 and 4 ("$\perp$" represents the absence of value) with respect to the second signal. SIGNAL uses relative *clocks* rather than a time index. Consider the following equation:

$$y_n = if\ x_n > 0\ then\ x_n\ else\ \perp \quad (1)$$

and the usual addition of sequences, namely

$$z_n = y_n + u_n \quad (2)$$

In combining these two equation, it is certainly preferable to match the successive occurences $y_1, y_2, ...$ in (2) with the corresponding present occurences in (1). But this is in contradiction with the immediate mathematical interpretation of the system of equations (formed by (1) and (2)), which yields $z_n = \perp + u_n$ whenever $x_n \leq 0$, and certainly does not match the usual interpretation of the addition of sequences. So, the use of such a notation is not convenient either as a specification technique or as a mathematical model. The SIGNAL language uses "clocks", the example bellow shows a signal and its clock (observed relatively to a given context) :

X	:	x_1	x_2	$\perp$	x_3	$\perp$	x_4	x_5	x_6	...
^X	:	T	T	$\perp$	T	$\perp$	T	T	T	...

The ^X signal is of *event* type (a boolean always true) and is present when X is present and absent otherwise. It represents the clock extraction.

Any two signals which are absent ($\perp$) simultaneously for any environment are said to be *synchronous* (e.g a signal and its clock are synchronous).

2.2 SIGNAL Kernel

A SIGNAL process is a set of relations between signals, specifying both constraints on values and constraints on relative timing of these signals (i.e constraints on clocks). The SIGNAL kernel is the minimum set of operators with which we can construct any SIGNAL process.

- **Functions:** Usual functions AND, OR, +, *, etc from instantaneous domains (boolean, integer,...) are extended to signals. Y $:= f$(X1,...,Xn) where Y,X1,...,Xn, are synchronous.

- **Delay:** The delay operator gives access to past values of a signal. The SIGNAL statement corresponding to delay is: ZX := X $ 1 , with ZX **init** x_0. For every occurrence of the signal X, ZX carries the previous value of X. We can easily extend this operator to a delay of k where k > 1.

- **When:** This operator allows one to conditionally extract values from a given signal. The expression: Y := X **when** C, where X and Y are signals of the same type and C is a boolean signal describing the under sampling of X. Y is present when both X and C are present and C is TRUE.

X :	x_1	x_2	$\perp$	x_3	$\perp$	x_4	x_5	x_6	...
C :	F	T	F	F	T	T	$\perp$	T	...
Y :	$\perp$	x_2	$\perp$	$\perp$	$\perp$	x_4	$\perp$	x_6	...

- **Default:** This operator allows the merge of signals of the same type: Z := X **default** Y, Z merges X and Y, with priority to X when both signals are simultaneously present.

X :	x_1	x_2	$\perp$	x_3	$\perp$	...
Y :	y_1	$\perp$	y_2	$\perp$	y_3	...
Z :	x_1	x_2	y_2	x_3	y_3	...

2.3 Processes Composition And Extended Language

We can build elementary processes as equations between signals " X := *exp*", where X is a signal and *exp* is any valid SIGNAL expression built using the kernel operators and other signals (e.g X:= (A when B) default (C+D)). We can also express equations as constraints on signals :" **synchro** { *exp1, exp2, .., expn* }" means that *exp1,..,expn* must be synchronous. More complex processes can be built using the commutative and associative composition operator "|". The process

(| P1 | P2 ....| Pn |)

simply denotes the union of the constraints expressed by P1,P2,...,Pn, where Pi is an elementary process (either a signal definition X:= exp or a synchro statement) or a composed process.

Consider the following example :

```
(| Y := X default ZY
 | ZY := Y$ 1
 | synchro { Y, ^X default when B)} |)
```

The first statements means that Y is equal to X when X is present and equal to the value of ZY (Y's last value, see the second statement) when X is absent and B is true (specified by the synchro statement). This is a memory operator (see example below).

X :	x_1	x_2	$\perp$	x_3	$\perp$	x_4	$\perp$	$\perp$	...
B :	T	$\perp$	T	F	T	F	T	F	...
Y :	x_1	x_2	x_2	x_3	x_3	x_4	x_4	$\perp$	...

Other operators have been defined from the kernel that permit the reduction of programming effort. The memory operator described in the previous example is noted as follows : Y := X **cell** B.

SIGNAL offers the possibility to specify constraints on clocks : $A\hat{} = B$ denotes the equality of clocks, i.e. A and B are synchronous. $A \hat{}+ B$ denotes the upper bound or the union of clocks, i.e. it is the clock present when A or B are present. Note that all the clock operators described above can be expressed using the SIGNAL kernel. Other operators and array based operators can be found in (Le Guernic 1991).

2.4 Compiler And Proof System

The SIGNAL compiler transforms any program into the kernel language, and tries to verify constraints implied by signal assignment statement (recall that any signal assignment statement sets constraints on data and clocks) or explicitly specified by the constraints-specification capabilities.

The checking of the coherence of constraints is done by the compiler. The compiler is in fact a formal calculus system. It handles (i) the presence/absence (ii) boolean values (iii) dependency graphs to encode data dependencies in non-boolean functions. This is called the clock calculus. If an incoherence is detected the program is rejected, in the case where the system constraints are coherent there are two possibilities: (1) if the system is fully functional then it generates an execution schema (2) the system is relational i.e it sets constraints on its inputs. In the second case we can not always have an execution schema but we can prove properties on the partially described system.

The formal proof is done by transforming the dependency graph and the constraints of clocks into a more compact representation:Ternary Decision Diagram *TDD*, an extension of the Binary Decision Diagram (Dutertre 1992).

2.5 The SIGNAL Environment

The SIGNAL environment consists of: a mixed schematic/textual programs entry, a compiler, and generation tools (see Figure 1). The compiler transforms SIGNAL programs into an intermediate graph format. This format is used for sequential code generation (C, Fortran77). It can be also used for parallel execution on general purpose machines: iPSC, T-node (some details and applications on these architectures are presented in e.g. (ICS 1990)) or specialized parallel machines. The intermediate format is also used for generation of *TDD* and for hardware synthesis.

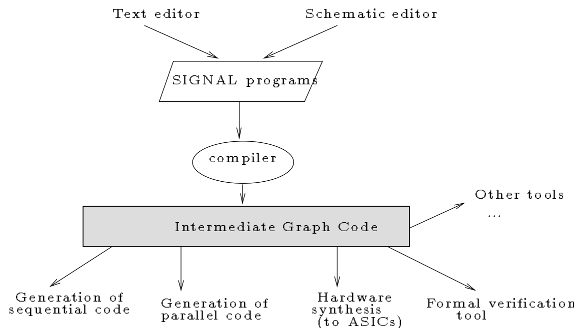


Figure 1 : SIGNAL environment

3 WHY USING VHDL AND SIGNAL ?

From our point of view, SIGNAL is used as high level design tool on top of VHDL (see Figure 2). The use of VHDL is motivated by several reasons. One of them is the access to existing CAD/CAE tools: as VHDL is the standard HDL, compiling SIGNAL into VHDL gives possibilities to use tools developed around it.

Another important reason is the simulation capabilities of both languages: SIGNAL always computes the current output using current and/or past values of the input, while VHDL has a preemptive simulation model that computes the possible future output using actual inputs. VHDL uses an event-driven simulation that leads to good performance when modeling hardware at the gate level with detailed timing information, while SIGNAL will lead to an inefficient simulation. Consider the following tiny program:

```
process INV_GATE= (integer DELAY)
        {? logical X  ! logical Y}
        (| Y := not (X $ DELAY) |)
end
```

If we consider that DELAY represents the propagation time of INV_GATE expressed in nano-second (in SIGNAL this means: X and Y are synchronous with a clock that ticks every ns). Suppose that the input X changes every micro-second. In SIGNAL the output is computed every ns, while in VHDL we will have two transactions every micro-second. SIGNAL cycle-driven simulation mode will be efficient when describing clocked logic or event-based circuit, it seems to be inefficient if detailed timing information is modeled.

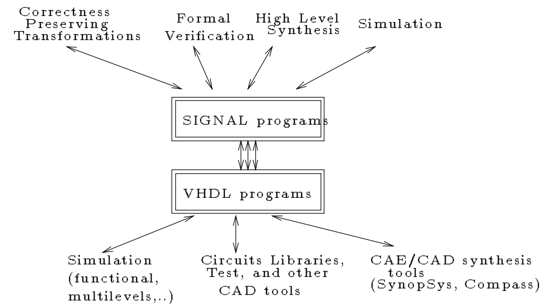


Figure 2 : VHDL/SIGNAL interface

Thus, when the hardware modeling is at low or mixed (low/high) levels, it is suitable to use VHDL simulator, while the high level description (Register-transfer, behavioral) can be handled by SIGNAL.

This cooperative approach (combining the two languages) can be extended to other activities than simulation: use of formal proof capabilities of SIGNAL for some VHDL programs, incorporation of VHDL libraries in SIGNAL designs, using time abstraction and hardware/software codesign mechanisms of SIGNAL, and capability of describing partial specifications in SIGNAL (suitable for system level design where the designer does not necessarily know the exact functionality of his sub-systems).

The next section describes how we intend to build the interface to handle the activities listed above.

4 INTERFACING VHDL AND SIGNAL

VHDL has been defined with a simulation oriented behavior. It consists of evaluating every simulation step if there is at least one process to execute (we suppose that a VHDL program is a set of processes). A process can be executed if the **wait** statement can be passed. If no process can be executed, move to next value of TIME (where a new value of a signal may pass its wait statement(s)).

The SIGNAL language does not use the same simulation model. No reference to the future is allowed. Execution time is always incremented by one (no "jump" is done) and physical time is not used.

It is obvious that the two styles of simulation model serve different needs. We will present how we can link SIGNAL model with VHDL.

This section is divided into four parts. The first one discusses the representation of signals in both languages. The second and third ones present VHDL to SIGNAL and SIGNAL to VHDL interfaces, and finally the last part discusses the limitations on the subsets of both languages that can be translated.

4.1 The Representation Of Signals

One of the most important features of both languages is the representation of signals. However, the signals do not have the same representation and modeling in the two languages.

In SIGNAL, a signal is a flow of values using logical time "*clocks*" (cf. section 2). No absolute time exists for signals. A signal is observable *only* when it is present (its clock is true).

In VHDL, a signal is a flow of values using a time metric (e.g femto-second), and holds its value (i.e. it is observable) between two events.

Thus, the first step is to represent VHDL's time model in SIGNAL and conversely (SIGNAL in VHDL).

4.1.1 Modeling of VHDL timing in SIGNAL VHDL uses an event-driven simulation-oriented semantics. The semantics of its timing constructs are expressed as operations on an abstract discrete event simulator. In this section we give a model for time scales in VHDL.

• Time scales VHDL uses two scales of time: the real or macro-time, and the delta or micro-time (Figure 3). The macro-time scale represents "real" time, e.g. a nano-second or a micro-second. The smallest unit of time which can be expressed in VHDL is the femto-second.

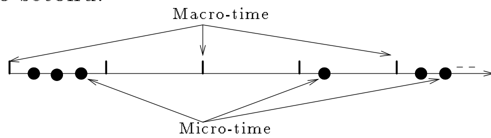


Figure 3 : Time scales

The micro-time (or delta time) is a non-measurable delay. Any (even infinite) number of delta's can exist between two instants of real time.

• The modeling of time scales Unlike the model described in (Pierre *et al.* 1992; Wilsey 1992), the model described here uses both macro-time and micro-time.

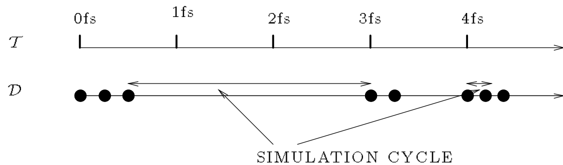
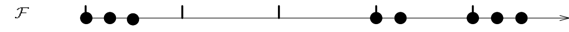


Figure 4 : Time models

Let the clock $\mathcal{T}$ represent real time (or macro-time). Let the clock $\mathcal{D}$ represent micro time, where every step corresponds to one simulation cycle (see figure 4).

Let $\mathcal{F}$ denote an imaginary global clock. It corresponds to the union of $\mathcal{T}$ clock and $\mathcal{D}$ clock. We can express this formally in SIGNAL with the following clock relation: $\mathcal{F}\hat{}=\mathcal{T}\hat{}+\mathcal{D}$ corresponds to $\mathcal{F} = \mathcal{T} \cup \mathcal{D}$.

A possible graphical interpretation is shown below:



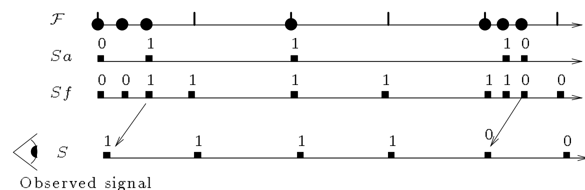
The VHDL current simulation time denoted by NOW, can be easily described in SIGNAL as: NOW := # $\mathcal{T}$-1, where '#' counts the number of $\mathcal{T}$ events.

Signals are visible to the VHDL user in macro-time, i.e. in the domain of $\mathcal{T}$, and it is in macro time that signals are available for interactions with exterior processes. A signal may not be observed to change faster than macro time. Nevertheless, the signal can be represented in micro-time during calculations.

• Signal Modelization One can identify two different models for a VHDL signal S, depending on the time domain in which the signal is used.

For the simulation domain, we use the representation *Sa*, which is the value of S when it is active.

For the observation domain (the external user's view) the signal is noted *S*. The observed (or real-time) signal can be derived from simulation domain across imaginary clock $\mathcal{F}$ (*Sf*=*Sa* **cell** $\mathcal{F}$), by down-sampling *Sf*. A possible pictoral view of the signal models is given below.

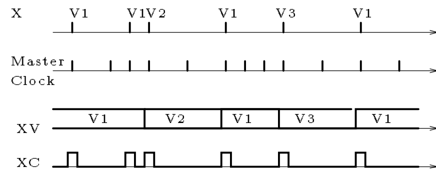


Note that if no zero-delay signal assignement is used the VHDL signals can be represented with $\mathcal{T}$.

• Variable Modelisation Variables in VHDL are represented in SIGNAL as *signals*. The SIGNAL assignment operation is instantaneous, it is the same as the delay semantics of variables in VHDL. We suppose that variable representation in SIGNAL is synchronous with respect to $\mathcal{D}$.

4.1.2 Modeling of SIGNAL timing in VHDL For executable SIGNAL programs, the compiler synthesizes a master (logical) clock. Every signal can be described as a down-sampling of the master clock. Thus, for every signal X, we have two corresponding VHDL signals : XV, XC. XC is true when X is present and false otherwise. XV has the value of X when XC is

true otherwise its value is not important (we keep the same value).



Note that if all Input/Output signals are synchronous we need only one signal XV for a signal X (the clock signals are not needed).

4.2 Translating VHDL Into SIGNAL

We use a small VHDL subset for validating the model; it is composed of: processes that contain one **wait** before the key word **end process**, ports of mode **in** and **out**, scalar type signals and variables, variables and signal assignment and the **If-then-else** control structure.

4.2.1 Signal attributes Due to space limitations, we will only present the EVENT attribute; for the remaining attributes see (Belhadj *et al.* 1993).

The EVENT attribute is expressed by taking advantage of the micro-time capabilities of SIGNAL. Following the definition, EVENT is true when there is a change in the value of the signal:

```
process EVENT=          % process name %
 {? logical SA;         % input of logical type %
    event CLK_D         % input clock D %
  ! logical EVENT }     % output EVENT %
  (| EVENT:= (when(not(SA=ZSA))default (not CLK_D)
   | ZSA:= SA $1 |)
where
 logical ZSA init false %local signal declaration%
end;
```

4.2.2 Variable assignment As the variables propagate with no delay, variable assignment can be considered as a SIGNAL assignment. We substitute the value of variables with the corresponding expression. For example consider the variable V defined in following the process:

```
 variable V: ...;
 begin
 V := EXP1;
 use of V;
 ...
 V := EXP2;
 use of V;
```

For the first use of V we substitue the value of EXP1, and EXP2 for the second one. If a variable is used before it has been assigned then V$1 is used (variables hold their old value) for all occurence before the V assignment.

4.2.3 Signal assignment statement In SIGNAL no reference to future is allowed, the statement Y <= **transport** X **after** N, is translated as Y <= X'DELAYED(N) which can be coded in SIGNAL as :

```
process TRANSPORT_DELAY=
         (integer N)
         {? X; event CLK_T
          ! Y}
        (| synchro {X,Y,CLK_T}
         | Y := X$N |)
end
```

For the inertial delay, the input wave form has to be longer than delay N to appear in the output. We use the process STABLE(N){X} that is true when the number of time ticks during which the signal X did not change is greater than N.

```
process INERTIAL_DELAY=
   (integer N)
   {? X; event CLK_T
    ! Y}
   (| Y  := (X$N) when STABLE(N){X} default ZY
    | ZY := Y$1
    | synchro{X,Y,CLK_T} |)
  ...
```

For signal assignment without delay, as the model captures micro-time, it can be coded as follow:

```
 Ya := Xa$1
```

4.2.4 If-then-else statement For the conditional structure, we use the following schema :

```
IF C THEN  S1 <= EXP1;
     ELSE  S1 <= EXP2;
ENDIF;
```

It is transformed into the SIGNAL statement :

```
 S1 := EXP1 when C default EXP2 when not C.
```

This can be extended to the case form.

4.2.5 Wait statement Let's consider the statement:

```
 wait on S1,S2 until B for TEXP;
```

We write a SIGNAL process that evaluates true if the wait is passed (i.e if either B is true when there is an EVENT on S1 or S2, or the process is waiting for TEXP units of time). The corresponding SIGNAL program is:

```
process WAITING=
   (integer TEXP)
   {? S1;S2;
      logical B;
      event CLK_T
    ! logical Y}
   (| Y    := when(START)
           default WATCH(TEXP){START,CLK_T}
    | START:= ( when (EVENT{S1,CLK_T}) default
           when (EVENT{S2,CLK_T})) when B |)
.... % intially START is true
```

where WATCH(TEXP){START,CLK_T} is true whenever the process is waiting for TEXP units of time (CLK_T) after the last occurence of START.

4.2.6 Process statement Let's consider that the VHDL program contains m processes $P_0, .., P_m$.We transform the sequential VHDL processes into SIGNAL parallel (data flow) processes.

Thus, successive execution of a VHDL process (simulation step) is done by the **wait** activation (note here Wi is the waiting statement of process Pi). When Wi is true the process Pi is executed.

For the signals in the input/output interface we use the SIGNAL format: we represent the value of a signal for every time unit of $\mathcal{T}$. Then, we use the following rules for the generation of clock $\mathcal{D}$ (simulation step) : if W_0 or...or W_m then a new simulation step ($\mathcal{D}$) is generated, otherwise move to the next step of $\mathcal{T}$.

4.3 Translating SIGNAL Into VHDL

This step is simpler than the VHDL to SIGNAL interface. We consider only the functional (executable) SIGNAL programs. Then, two generators can be built: a behavioral and a structural one.

Behavioral translation uses the same scheme as the one used for sequential code generation (C and Fortran77). SIGNAL programs are equational, it is the compiler who generates a control structure. The generated code may contain conditionals (If-then-else), loops, variables and signal assignments in one VHDL sequential process. We use the VHDL representation of SIGNAL signals discussed previously.

Structural translation converts graphic interface information, boxes (representing processes) and interconnexions (signals) into structural VHDL.

4.4 Constraints On The Interface

Translation in both directions can be extended, but we must keep in mind some important issues:

- The first constraint of an interface is that simulation of VHDL or SIGNAL programs obtained by translation must be equivalent.

- We can translate non executable SIGNAL processes into **assert** statements and generate a VHDL entity without architecture, but the SIGNAL to VHDL interface is designed to be used as CAE/CAD entry. Thus, it must use functional (executable) programs.

- Some limitations on the data types come from both sides. For VHDL, as we intended to generate synthesizable VHDL code, e.g. real types are not used because they cannot be handled by current synthesis tools. Some data types in VHDL do not exist in SIGNAL's current version.

- Not all VHDL is translatable in SIGNAL. Moreover, the possibly translatable VHDL is not always useful for formal verification. This is due to the current limitation of the proof system of SIGNAL (e.g. integer based proofs).

5 CONCLUSION AND FURTHER WORKS

We have presented a study and design of an interface between two languages (VHDL and SIGNAL) and their environments. The major difficulty was the different simulation models. We describe a common representation in both models. This will permit the simulation of SIGNAL programs in VHDL and VHDL programs in SIGNAL. As the capabilities and the tools of the two languages exist, the important aspect was the accuracy of the interface. However, we found a number of restrictions and limitations. We are now improving our interface by adding more constructs, to handle a larger set of designs.

This, we hope, will permit high level design using SIGNAL and VHDL, and easy design of Real-time and Signal processing systems (hardware and software).

Thanks to E. Rutten, T. Gautier, R. McConnell, D. Wilde and anonymous reviewers for their careful reading of the paper.

This work is partially supported by the Doctoral-candidate Network for System and Machine Architecture of the DRED

REFERENCES

- Belhadj M., R. McConnell and P. Le Guernic P. 1993. "A framework for macro- and micro-time to model VHDL attributes." In *Proceedings of the European Design Automation Conference*(Hamburg, Sep.).IEEE, 520-525.
- Dutertre B. 1992. *Spécification et preuve de systèmes dynamiques* Ph.D. thesis University of RENNES.
- ICS 1990. *International Conference on Supercomputing.* ACM Press.
- LRM 1987. *IEEE Standard VHDL Language Reference Manual.* IEEE Press. IEEE Std 1076-1987.
- Pierre L.; D. Borrione; and A. Salem. 1992. "Formal verification of VHDL description in the Prevail environment." *IEEE Design & Test of Computers*,(Jun.):42-55.
- Le Guernic P. 1991. "The signal programming environment." In *Proceedings of the Conference on Algorithms and Parallel VLSI Architectures II* (Gers, France, June). Elsevier Science Publishers B.V.,347-358.
- Wilsey P. A. 1992. "Developing a Formal Semantic Definition of VHDL." In *VHDL for simulation, Synthesis and Formal Proofs of Hardware*, J. Mermet ed. Kluwer Academic Publishers,245-256.