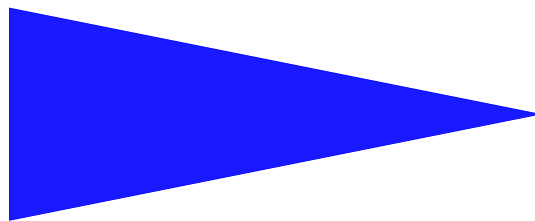
IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION INTERNE

N° 1131



A SYNCHRONOUS APPROACH FOR HARDWARE DESIGN

MICHEL ALLEMAND, FRANÇOIS BODIN, APOSTOLOS KOUNTOURIS,
PAUL LE GUERNIC, JEAN-CHRISTOPHE LE LANN,
ANDRÉ SEZNEC, CHRISTOPHE WOLINSKI

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

Institut de Recherche en Informatique et Systèmes Aléatoires
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

A Synchronous Approach for Hardware Design

Michel Allemand, François Bodin, Apostolos Kountouris,
Paul Le Guernic, Jean-Christophe Le Lann, André Seznec,
Christophe Wolinski

Thème 1 — Réseaux et systèmes
Projet CAPS,EP-ATR

Publication interne n ° 1131 — Octobre 1997 — 34 pages

Abstract: In this report we present a methodology for designing complex hardware systems. This methodology is based on the synchronous data flow language SIGNAL which offers a formal framework to build executable specifications of hardware components. All design steps (i.e. refinements, verification, simulation, HDL generation, ...) are based on this unique formalism which allows to reduce product design cycle by decreasing communication problems between design phases. In this report we emphasis on the verification process and the HDL generation. The methodology can be applied to the dataflow synchronous common format DC+ [19].

Key-words: Signal, synchronous data flow, HDL, hardware design, methodology

(Résumé : tsvp)

Centre National de la Recherche Scientifique
(UPRESSA 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Une approche synchrone pour la conception de matériel

Résumé : Ce rapport présente une méthodologie pour la conception de systèmes matériels complexes. Cette méthodologie est fondée sur le language flots de données synchrones Signal qui offre un cadre formel pour l'écriture de spécifications exécutables de composants matériels. Toutes les étapes (i.e. raffinement, vérification, simulation et génération HDL, ...) sont fondées sur cet unique formalisme ce qui permet de raccourcir le cycle de production en réduisant les problèmes de communications entre les phases de conception. Dans ce rapport nous mettons l'accent sur les processus de vérification et de génération de code HDL. La méthodologie s'applique aussi au format commun flots de données synchrones DC+ [19].

Mots clés : Signal, flot de données synchrone, conception de matériel, méthodologie

Introduction

In most approaches to complex hardware system design (i.e. microprocessors, micro-controllers ...) multiple formalisms such as natural languages (i.e. English), standard programming languages (i.e. C), hardware description languages (i.e. Verilog [23] or VHDL [24]) and specific formalisms are used in the distinct specification steps. One of the consequences of design methods built on multiple formalisms is that the product design and checking is inherently long. Typically, from a specification written in a natural language a simulation oriented description (usually in C) and a synthesis oriented implementation (usually as a RTL HDL program) are derived. Such design methodology results in a very long cycle production time since communications between phases are difficult and error prone. For instance, when a design error (or an unexpected timing constraint) is discovered during the implementation, updating the design chain is very time consuming and may result in further errors. Furthermore the use of formal methods is very difficult in such a multi-formalisms framework.

In this report, we present a design methodology for complex hardware systems where the synchronous language SIGNAL [17] is used as the unique formalism for all design steps. SIGNAL offers a very adequate formalism to hardware specification as it provides an implementation independent abstraction based on synchronous data flows.

One of the main advantage of SIGNAL is provided by its formal semantic that allows directly the application of formal verification methods. Correct by construction behavioral simulators and proved equivalent synthesizable HDL descriptions can also be automatically derived from a SIGNAL specification. An incremental development from a high level description to a very precise one is done through the specification of the implementation choices. These refinement steps are, as much as possible, proved to be correct and automatically performed.

This report presents a preliminary description of the methodology for hardware design based on SIGNAL. Section 1 overviews the SIGNAL language, its functional and operational semantics. Section 2 describes the methodology for designing and deriving an application from an **executable specification**. In Section 3, we show how synthesizable HDL is derived from a SIGNAL spe-

cification. In particular, we emphasis on the concept of well clocked implementations that defines how abstract signals are mapped to implementable ones. Section 4 provides an overview of the formal verification process based on model checking techniques.

1 Overview of the Signal Language

Signal [18] is a declarative real-time synchronous data-flow language. It allows to define an application as a relation between typed (mathematical) variables defined on sequences of values (the signals); these relations are described as a set of constraints built on signals as equations on domains including boolean, enumeration, integer, real, arrays and structures associated with a standard set of operators on those traditional data types.

1.1 Signal

Signal is built around a minimal kernel. It manipulates *signals* which are unbounded series of typed values with an implicitly associated *clock*. In this language the notion of physical time (chronometric) is replaced by a logical notion of time (order in a denumerable set of events and a "simultaneous" equivalence relation). The *clock* of a signal denote the set of instants (relatively to the other signals) where values are present. This notion is illustrated figure 1. The **absence** of a signal (relatively to the presence of other signals) is denoted by $\perp$.

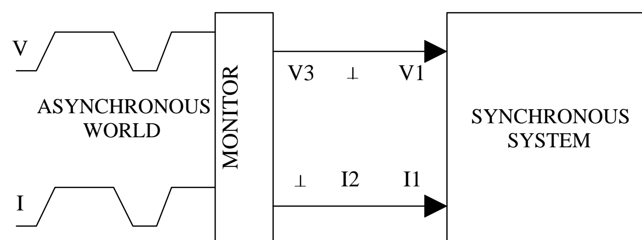


Figure 1: Asynchronous model versus the synchronous model.

Signals of a special type *event* are characterized only by their clock, i.e. their presence (they are given the boolean value *true* at each occurrence and $\perp$ else). Given a signal `X`, its clock is obtained by the language expression `event X`, resulting in the *event* signal that is present simultaneously with `X`.
The constructs of the language are used to specify the behavior of systems in an equational style: each equation states relations between signals (i.e., between their values and between their clocks). Systems of equations on signals are built using the composition construct. In this sense, it is a form of constraint-based programming. When equations define a function from input values to output values (ie. is a dataflow function), the resulting program is reactive (i.e., input-driven). In other cases (it is not enough to consider values, absence symbol is needed to get functions), correct programs can be demand-driven or control-driven. A program is said control/demand-driven if further relationship must be specified by the execution context to produce an input-driven program.

1.1.1 Kernel of SIGNAL

The kernel includes the five following constructs:

- **Functions:** `Y := R(X1,X2, ...,Xn)` (e.g., addition, multiplication, conjunction,...). They are defined on the types of the language. For example, the equation stating that signal `F` is the boolean negation of signal `E` is written `F := not E`.
  The signals `Y`, `X1`, ..., `Xn` must all be present at the same time. Those operators for which inputs and outputs have the same clock are called *monochonous operators*.
- **Delay operator ($):** `ZA := A $ 1` It is a monochonous operator which gives the previous value `ZA` of a signal `A` relatively to the clock of `A`. An initial value has to be given to `ZA`. It is the only way to access past values of signals. Signals `A` and `ZA` have the same clock. By definition `A $ (n + 1) = (A $ n) $ 1` when `n` $>$ `0`. The `$` operator is illustrated figure 2.
- **Extraction operator:** `Y := X when C` . The values of `Y` are produced by extracting the values of `X` when the values of the boolean condition `C` are present and true. The operands and the result do not have identical

```
| ZA := A $ 1
| ZB := A $ 1
| ZC := A $ 3
where
integer ZA, ZB init 0,
ZC init [[1]:10, [2]:20,[3]:30]
```

A	1	2	⊥	3	4	5	⊥	6
ZA	?	1	⊥	2	3	4	⊥	5
ZB	0	1	⊥	2	3	4	⊥	5
ZC	10	20	⊥	30	1	2	⊥	3

Figure 2: `$` operator example. Note that `ZA` and `ZB` have the same clock as `A`.

clock, `when` is then called a *polychronous operator*. The clock of signal `Y` is the intersection (a clock can be seen as a set of instants) of the clock of `X` and the clock of occurrences of `C` at the value *true*. This operator is illustrated figure 3.

```
          ok := x <= 1 when found
x         1  0     2  3  0     3
found        t        t  f
ok           t
```

Figure 3: `when` operator example.

- **Merge operator:** `S := A default B`. The deterministic merge is a polychronous operator which defines the union of two signals of the same type, with a priority on the left one when both are present simultaneously. The clock of `S` is the union of that of `A` and that of `B`. `S` holds the value of `A` when `A` is present, otherwise `S` is equal to `B`. This operator is illustrated figure 4.

- **Parallel composition** of processes is an associative and commutative operator "`|`", denoting the union of the underlying systems of equations and thus the intersection of the solutions. Systems communicate and interact through signals defined in one system and featured in others. For

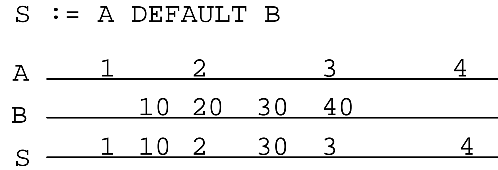


Figure 4: **`default`** operator example. Clock of `S` is the union of clocks of `A` and `B`.

> these signals, composition preserves constraints from all systems, especially temporal ones. This means that they are present if the equations systems allow it. In SIGNAL, for processes `P1` and `P2`, composition is written: `P1 | P2`

Except for the local declarations that need the introduction of the notion of restriction (see [4]), the rest of the language is built upon this kernel. A structuring mechanism is proposed in the form of process schemes, defined by a name, typed parameters, input and output signals, a body, and local declarations. Occurrences of process schemes in a program are expanded (like macro-expansion) by a pre-processor of the compiler. Derived processes have been defined from the primitive operators, providing programming comfort: e.g., **`synchro{X,Y}`** which constrains signals `X` and `Y` to be synchronous; **`when`** `C` giving the clock of occurrences of `C` holding **`true`**; `X` **`cell`** `B` which memorizes values of `X` and outputs them also when `B` is *true*. Arrays of signals and of processes have been introduced as well. For a complete description of SIGNAL see [4].

1.2 Overview of the Semantic of SIGNAL

In this section we briefly present the trace and operational semantics of SIGNAL. The trace semantic defines operations on sequences of values of set of signals. More details on these notions can be found in [16] and [15].

1.2.1 Trace Semantics: Flows, Signals and Clocks

Consider an alphabet (finite set) A of typed variables called *ports*. For each $a \in A, \mathcal{D}_a$ is the domain of values (integers, reals, booleans...) that may be carried by a at every instant:

$$\mathcal{D}_A = \cup_{a \in A} (\mathcal{D}_a \cup \{\perp\})$$

where the additional symbol $\perp$ denotes the absence of the value associated with a port at a given instant. For two sets A and B, the notation $A \rightarrow B$ will denote the set of all maps defined from A into B. Using this notation, we introduce the following objects:

- **Events.** Events specify the values carried by a set of ports at a considered instant. The set of the A-events (or "events" for short when no confusion is likely to occur) is defined as

  $$\mathcal{E}_A = A \rightarrow \mathcal{D}_A$$

  Events will be generally denoted by ϵ and their domain by $\mathcal{D}(\epsilon)$. We shall denote by $\perp_A$ the "silent" event ϵ such that $\epsilon(a) = \perp, \ \forall \ a \in \mathcal{D}(\epsilon)$.

a1	1	2	⊥	3	4	5	⊥	6
a2	⊥	1	⊥	2	3	⊥	⊥	5
a3	0	1	⊥	2	⊥	4	⊥	5
a4	10	20	⊥	30	1	2	⊥	3

Figure 5: Example of Trace.

- **Traces.** Traces are infinite sequences of events. Let $I\!N_+ = \{1, 2, \cdots\}$ denote the set of positive integers, then the set of A-*traces* (or simply "traces") is defined as

  $$\mathcal{T}_A = I\!N_+ \rightarrow \mathcal{E}_A$$

  An example of trace is given figure 5.

- **Compression.** The *compression* of an A-trace T (deleting the silent events) is defined as the (unique) A-trace S such that :

  $$S_n = T_{k_n}$$

  where $k_0 = \min \{m \geq 1 \ : \ T_m \neq \perp\}, \ k_n = \min \{m > k_{n-1} \ : \ T_m \neq \perp\}$

where $min\ \emptyset = +\infty$ by convention. The compression of a trace T will be denoted by $T\downarrow$.

- **Flows and signals.** The condition
$$T\downarrow = T'\downarrow$$
defines an equivalence relation on traces we shall denote by $T \sim T'$. The corresponding equivalence classes are called *flows*. The set of all possible flows on A will be denoted by $\mathcal{F}_A$, so that we have[1]
$$\mathcal{F}_A = (\mathcal{T}_A)_{/\sim}$$
Elements of $\mathcal{F}_A$ will be generically denoted by F_A or simply F when no confusion can occur. While the notion of trace refers to a particular environment (since the $\perp$'s are explicitly listed), the notion of flow does not. Since
$$\mathcal{F}_A = [\mathbb{N}_+ \rightarrow (A \rightarrow \mathcal{D}_A)]_{/\sim}$$
any $F_A \in \mathcal{F}_A$ may be written as
$$F_A = (F_a)_{a\in A}$$
and the F_a's are termed *signals*. Hence a signal is a component of a flow specified by selecting a particular port in the alphabet A. The flow corresponding to the trace figure 5 is shown figure 6.

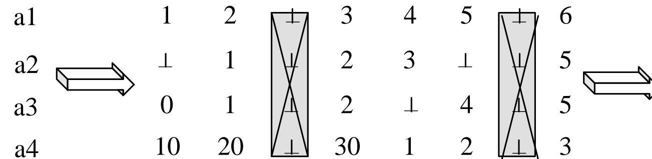


Figure 6: Example of Flow.

Definition of a Signal program P P is simply a subset, $\mathcal{F} \subset \mathcal{F}_A$, of the set of all flows on A. In other words, we consider a Signal program, as a way to specify "legal" flows. This set of flows is denoted $[\![P]\!]$ (i.e. interpretation of P) when P denotes the syntaxical description.

[1] $./\sim$ denotes here the quotient space by the relation $\sim$

Restricting P: Consider a subset A' of the alphabet A. The inclusion $A' \subset A$ induces a projection from $\mathcal{E}_A$ onto $\mathcal{E}_{A'}$. The trace restriction is defined as:

$$T \longmapsto T_{||A'}$$
$$\forall\ a,\ ((a \in A') \Longrightarrow (\forall\ t,\ (T_{||A'})_t(a) = T_t(a))$$

The restriction, $T_{||\{a2,a3\}}$, is illustrated figure 7. This notion is naturally extended to processes.

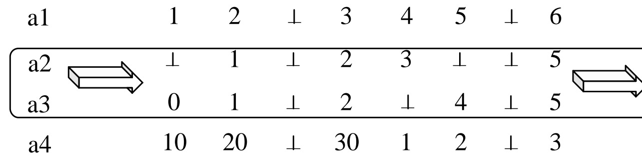


Figure 7: Trace restriction.

Process Composition: the composition of two processes P_1, on A_1 and P_2, on A_2, denoted by $P = P_1|P_2$ is the maximal flow set defined by:

$$P = \{F\ \in\ \mathcal{F}_{A_1 \cup A_2}\ /\ ((F_{||A_1} \downarrow\ \in\ P1) \wedge ((F_{||A_2} \downarrow\ \in\ P2)\}$$

This is illustrated figure 8.

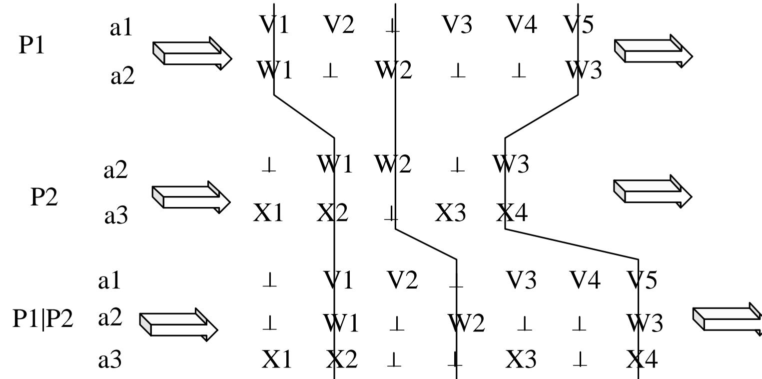


Figure 8: Process Composition.

Let us now define the semantic of the four kind of kernel operators introduced in section 1.1.1 (functions, delay, extraction and merging).

Functions: `Y := R (X1,...,Xp)`

$$\forall n \in I\!N_+,$$
$$\begin{array}{ll} & ((\forall i)\ Xi_n = \bot \wedge Y_n = \bot) \\ \vee & ((\forall i)\ Xi_n \neq \bot \wedge Y_n \neq \bot \wedge Y_n = [\![R]\!](X1_n, \cdots, Xp_n)\ ) \end{array}$$

Where **R** denotes a monochronous operator and $[\![R]\!]$ its interpretation. The notation Xi_n denotes the value carried by the port with name Xi at the n-th instant of the considered trace. This notation will be further used in the remainder of this section.

Delay: `Y := X$1 init x0`

$$\forall n \in I\!N_+,$$
$$\begin{array}{ll} & (X_n = \bot \wedge Y_n = \bot) \\ \vee & ((n > 1) \wedge Y_n = X_{n-1}) \\ \vee & ((n = 1) \wedge X_n \neq \bot \wedge Y_n = x0) \end{array}$$

Extraction: `Y := X when B`

$$\forall n \in I\!N_+,$$
$$\begin{array}{ll} & (B_n = true \wedge Y_n = X_n) \\ \vee & (B_n \neq true \wedge Y_n = \bot) \end{array}$$

Merging: `Y := U default V`

$$\forall n \in I\!N_+,$$
$$\begin{array}{ll} & (U_n \neq \bot \wedge Y_n = U_n) \\ \vee & (U_n = \bot \wedge Y_n = V_n) \end{array}$$

2 Methodology for Synchronous Hardware Design

Figure 9 illustrates the methodology we propose for synchronous hardware design. It is based on refinements to achieve the final specification. We can

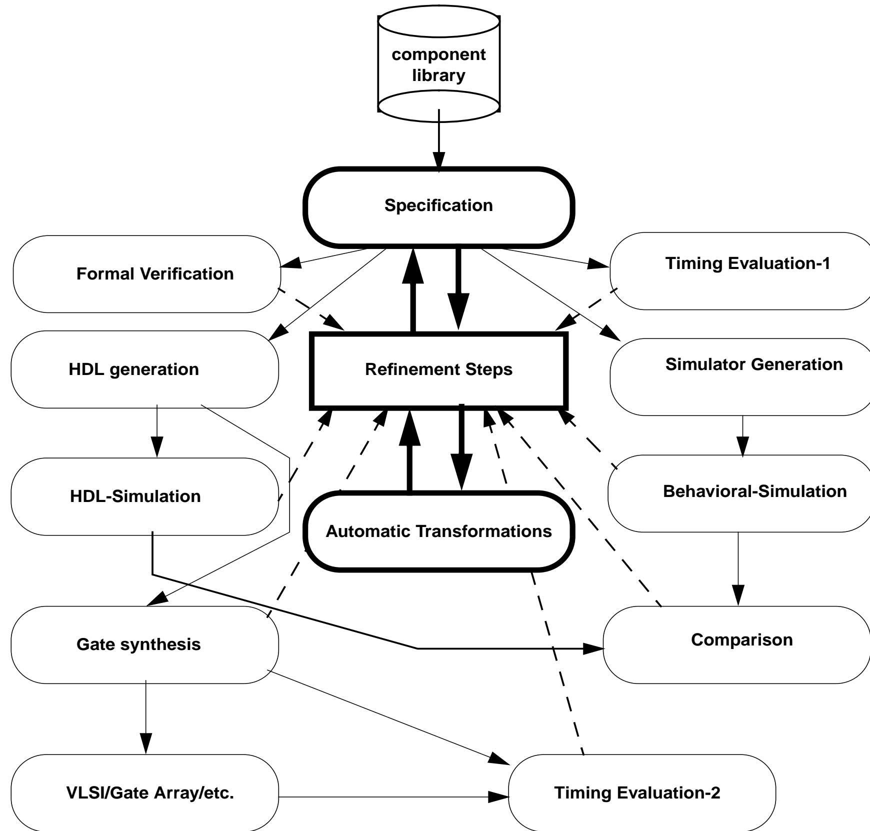


Figure 9: Methodology for hardware specification.

distinguish two kinds of refinements: **behavioral refinements** which consist in modifications of the specification and **implementation refinements** which consist in adding the implementation details to the specification. Refinements may be done manually but also automatically (correctness is ensured in this case). Modification to the specification are made according to the information provided via various tools which are applied on the specifications:

Formal Verification: The first verification is provided by the compilation of the specification (see section 4.1 about the compilation process). Then other verification methods (model checking, theorem provers) are applied so properties of the specification are formally checked. This step consists

essentially in two phases. In phase one, the formal properties to be verified by the component description are given. The second phase, mostly automatic, is the verification of the properties.

Component Library: Hardware components libraries can be taken into account into this process assuming that interface constraints and a behavioral model are provided. At any step in the design, the user can integrate a predefined library components.

Behavioral/HDL Simulations: Simulators can be provided either from the SIGNAL specification (via C code generation) or from the derived HDL description. As SIGNAL signals are comparable to HDL signals, the simulation results can be compared directly. Deriving the simulator directly from the SIGNAL specification can be done at any step of the design and allows to get, generally, faster simulation.

Timing Evaluations: The timing evaluations provide basic data to choose the implementation of the hardware components [12].

HDL Generation: When a SIGNAL description is a complete specification of a hardware component (i.e. input-driven, it has been successfully processed by the SIGNAL compiler), then it can always be fully compiled in some HDL language. However we are assuming that gate synthesis is performed through third party tools such as Synopsys.

```
process INT_MUX2=
{  ?  logical CTR; %CTR, V1, V2 are input signals%
   integer V1;
   integer V2
   !  integer VAL } %VAL is the output signal%
(| VAL := (V1 when (not CTR)) default (V2 when CTR)|)
end
```

Figure 10: Description of an integer multiplexer in SIGNAL.

Writing the specification of a component is performed in a top-down manner consisting first in the mapping of the input/output signals to SIGNAL and then in describing the relationship between these signals. For each signal, a type must be chosen to carry the data. In SIGNAL, among others the following types can be used:

event: An event can be used each time a signal has to trigger an action. For instance, any boolean signal active on true or false can be represented by an event.

logical: Any signal that must carry a logical value. For instance the signal command of a two entry multiplexer (shown figure 10).

integer: Most of the usage of integers is done as a preliminary step for any multi-bit values. As the specification is refined, integers can be replaced by a simpler types at the HDL implementation level.

arrays: arrays can be used to group signals, of the same type, if there are always used synchronously.

The component body is described as a set of operations (intrinsically a set of equations) on input signals (`?`) to produce the output signals (`!`). Figure 11 shows a specification of a simple resetable incrementer-decrementer counter. A second example is the description a simple finite state machine shown figure 13. It is described in SIGNAL using two integer signals, `S` denoting the next state, and `SZ` the current state. For each transition `T` a signal is created. This signal is an event that triggers the transition. For complex FSMs we provide a preprocessor that allows to write automaton description in a more classical style.

One of the key for refinements is to allow to check that a refinement produces a specification that is "equivalent" to the previous one. In SIGNAL this is simply done, either because the refinement is automatically performed and proven correct (such as the transformation used for HDL generation, see section 3), or using formal verification techniques such as model checking (see section 4). For instance figure 12 shows how equivalence can be checked of a component part P and the refined one P'. E_1, E_2, E_1' and E_2' allow to express equivalence between I/O signals.

```
process COUNTER =
{   ?  event RESET,CLK,DEC,INC
    !  integer I }
(| LNI := ((0 when RESET)
          default (LI when INC when DEC)
          default ((LI+1) when INC)
          default ((LI-1) when DEC)
          default LI) cell CLK
   | LI:= LNI$1
   | I := LI
|)
where
   integer LI init 0, LNI init 0
end
```

Figure 11: Incr.-Decr. Counter in Signal

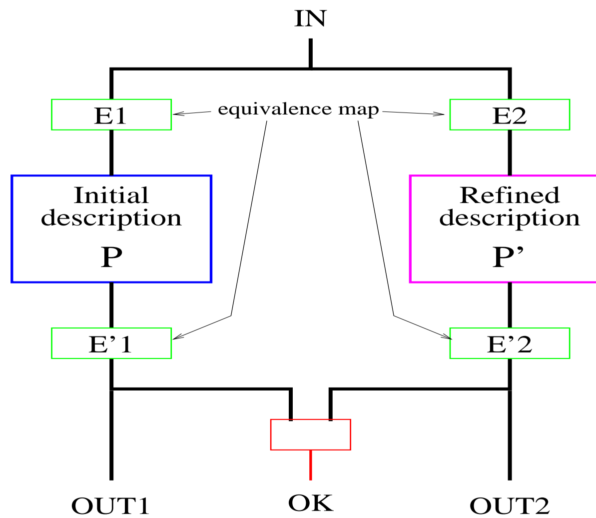
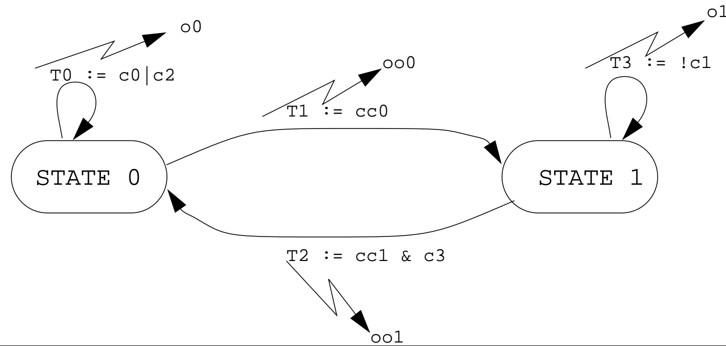


Figure 12: Refinement verification.

3 HDL Code Generation

We provide here a systematic, non optimal translation path from SIGNAL specifications to HDL descriptions. The most fundamental requirement for




```
process FSM=
{
   ?  event c0,c1,c2,c3,cc0,cc1,CLK
   !  event o0,o1,oo0,oo
}
(| S := (1 when T1) default (0 when T2) default SZ
   | synchro S, CLK
   | SZ := S$1
   | T0 := (c0 default c2) when (SZ = 0)
   | T1 := cc0 when (SZ = 0)
   | T2 := (cc1 when c3) when (SZ = 1)
   | T3 := when ((not c1) default CLK) when (SZ = 1)
   | o0 := when T0
   | o1 := when T3
   | oo0:= when T1
   | oo1:= when T2
|)
where
   event T0,T1,T2,T3;
   integer S, SZ init 0
end
```

Figure 13: An automaton description in Signal.

the HDL code generation is that the resulting code should be functionally

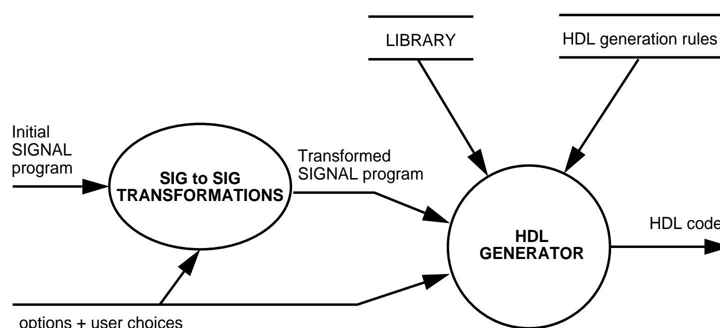


Figure 14: HDL generation process.

equivalent to the initial SIGNAL specification. In this section we first define the notion of implementation we use and then give a structural translation method, illustrated figure 14, to produce synthetisable HDL code.

3.1 Well Clocked Implementations

The well clocked implementation notion allows us to compare two SIGNAL processes, a specification P with a set A of I/O signals and an implementation P' with a greater or equal set A' of I/O signals. Roughly speaking, P' is a well clocked implementation of P if, the subset of I/O signals of P' that correspond to the I/O signals of P (a one to one correspondance σ) at each instant defined by a clock system of P (*definition of present*) hold respectively the same values. In other words, an implementation extends the event domain. Using strobe signals we are able to sample the events such that we get the initial specification.

A clock system of a process P defines a tree of signal clocks of P. It is mainly used to define the presence or absence of signals. More formally:

Definition 3.1 (Clock System) *Let P a process on A. A clock system on P is a function $s : A \longrightarrow A$ such that:*

- *if $c \in \mathcal{I}m(s)$ then c is a boolean.*
- *$\forall x \in A$ there exists an unique n (denoted $depth_s(x)$) such that*

- $\forall p \geq n, s^{p+1}(x) = s^p(x)$
- $\forall p < n, s^{p+1}(x) \neq s^p(x)$

- $\forall F \in [\![P]\!], \forall t \in I\!N_+, \forall x \in A,\ present(s)_t^F(x) = true \Longrightarrow F(t)(x) \neq \perp$
  where $present(s)_t^F(x) \triangleq \left(\bigwedge_{i=1}^{depth_s(x)} \{F(t)(s^i(x)) = true\} \right)$

Compiling a program P computes such an s clock system for P.

The sampling of a process P, according to a clock system s, consists in extracting the events selected according to s from the traces of P:

Definition 3.2 (Sampling) *Let P be a process on A and* s *a clock system on P (*s *can be produced by the compilation of P) such that $tick(P) = clk \in A$ (ie. clk is the fastest clock of P: $\forall x \in A, s^{depth_s(x)}(x) = clk$). Then we define the sampling function, denoted $\mathcal{S}_s$, by:*

$$F \in \mathcal{S}_s([\![P]\!])$$

$$\Longleftrightarrow$$

$$\left(\begin{array}{rl} \exists G \in [\![P]\!], \forall t, \forall a \in A, & present(s)_t^G(a) = true \Longrightarrow F(t)(a) = G(t)(a) \\ & \wedge\, present(s)_t^G(a) = false \Longrightarrow F(t)(a) = \perp \end{array} \right)$$

Using clock system and the sampling function, a well clocked implementation is defined as follow:

Definition 3.3 (Well Clocked Implementation) *Let P a process on A and Q a process on A', such that there exists a one to one correspondance σ such that $\sigma(A) \subseteq A'$, let* s *a clock system on Q.*
Q is a well clocked implementation of P with respect to s *(denoted $[\![Q]\!] \preccurlyeq_s [\![P]\!]$) iff:*

$$(\mathcal{S}_s([\![Q]\!]))_{||\sigma(A)} = [\![P]\!]$$

This definition means that in order to simplify the implementation we can add and rename signals of process P. In practice, that means that for each initial signal a we can add a strobed signal stb_a' and rename the initial signal into a', that both a' and stb_a' will be synchronized with the fastest clock clk, and if we look only at the signals a' of the new process, its behaviour is the same as the behaviour of initial process. Figure 15 illustrates the notion of well clocked implementation.

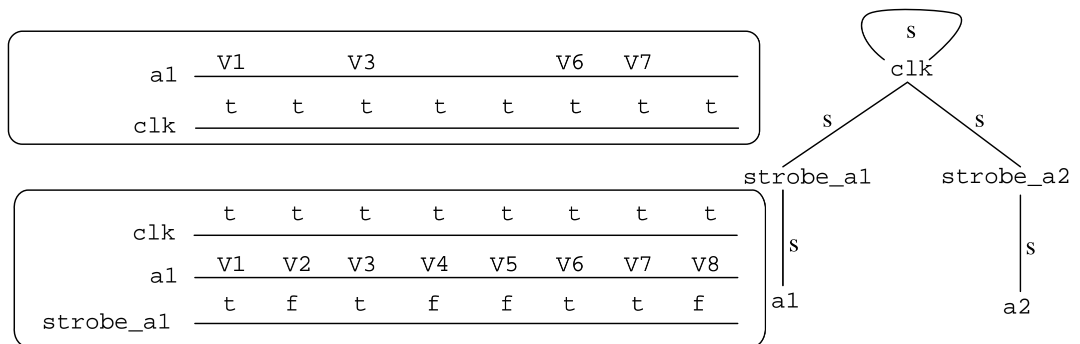


Figure 15: Well clocked implementation.

3.2 HDL Generation

The HDL generation is decomposed in two steps. A first transformation (I) modifies the SIGNAL specification in a new one that allows the HDL generation. This transformation must satisfy the property of well clocked implementation defined in previous section. The second step ($\mathcal{H}$) is a structural translation of the SIGNAL specification into the HDL target language (VHDL in our case). This is illustrated figure 16.

Figure 16: Transformations for HDL generation

3.2.1 The Strobe Insertion Transformation $\mathcal{I}$

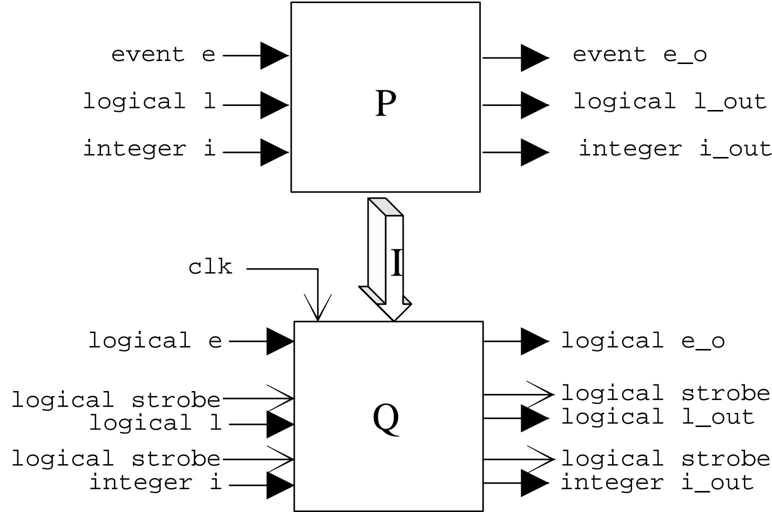


Figure 17: The strobe insertion transformation I.

The main goal of $\mathcal{I}$ is to transform the SIGNAL program in such a way that it is composed of signals that can be implemented. In our case this consists in transforming all signals into strobed signals that are synchronous to a signal `clk` that represents the circuit main clock. In other words all signals `a` are transformed in two new signals `(a', strb_a')`. This transformation is illustrated figure 17. More formally the transformation is defined as follow.

Definition 3.4 (*IN* and *OUT* sets) *Let P a process on A such that $clk \in A$ and $clk = tick(P)$ ($tick(P)$ is the fastest signal of P). Syntactically,*

- *$IN(P)$ is the set of inputs signals of P exept clk;*
- *$OUT(P)$ is the set of outputs signals of P.*

Definition 3.5 (Syntactical transformation $\mathcal{I}$) *Let P a process on A such that $clk \in A$ and $clk = tick(P)$, let σ and μ two one to one correspondances such that $\sigma(A) \cup \mu(A) = A'$ and $\sigma(clk) = \mu(clk) = clk$*

- $\forall a_i \in IN(P)$, *we note* $a'_i = \sigma(a_i)$ *and* $stb_a'_i = \mu(a_i)$
- $\forall b_j \in OUT(P)$, *we note* $b'_j = \sigma(b_j)$ *and* $stb_b'_j = \mu(b_j)$

We define $\mathcal{I}$ *(denoting* $\mathcal{I}_{\sigma,\mu}$*) by :*
$$\begin{array}{rcl} \mathcal{I} : \mathcal{P}_A & \longrightarrow & \mathcal{P}_{A'} \\ P & \longmapsto & Q \end{array}$$

such that:
$$\begin{array}{l} Q = (\mid <\ a_i := a'_i \ when\ stb_a'_i\ >_{i\in\{1..n\}} \\ \mid \{b_1, \cdots, b_m\} := P\{a_1, \cdots, a_n\} \\ \mid <\ stb_b'_j := true\ when\ event\ b_j\ default\ false\ >_{j\in\{1..m\}} \\ \mid <\ b'_j := b_j\ default\ b'_j\ >_{j\in\{1..m\}} \\ \mid <\ synchro\{clk, a'_i, stb_a'_i\}\ >_{i\in\{1..n\}} \\ \mid <\ synchro\{clk, b'_j, stb_b'_j\}\ >_{j\in\{1..m\}} \\ \mid)\ where\ a_1, \cdots, a_n, b_1, \cdots, b_m \end{array}$$

Where $< equation >_{k\in\{1..p\}}$ *means that we have an equation for all* k*, where* $n = Card(IN(P))$ *and* $m = Card(OUT(P))$.

Correctness of the $\mathcal{I}$ Transformation According to the definition (3.1) of a well clocked implementation, proving this transformation is done in two steps that we briefly illustrate in this section.

Process transformation: We prove first that the application of the transformation only on the external interface of a process is a well clocked implementation of the process. More formally:

Property 3.1 *Let* P *a process* A *such that* $clk \in A$ *and* $clk = tick(P)$ *(*$tick(P)$ *is the fastest clock of* P*) then*

$$[\![\, \mathcal{I}(P) \,]\!] \preccurlyeq_s [\![\, P \,]\!]$$

when s *defined by the following equations is a clock system of* $\mathcal{I}(P)$*:*

- $\forall a_i \in IN(P), s(a'_i) = stb_a'_i$ *and* $s(stb_a'_i) = clk$

- $\forall b_j \in OUT(P), s(b'_j) = stb_b'_j$ *and* $s(stb_b'_j) = clk$
- $s(clk) = clk$

The basic idea of the proof of this first transformation consists in showing that the two sets of traces $(\mathcal{S}_s([\![\mathcal{I}(P)]\!]))_{||\sigma(A)}$ and $[\![P]\!]$ are equal.

Composition of Transformations: The second part of the proof shows that the transformation of a composition of two processes is equivalent to the composition of the transformations of each process and thus that the recursive application of the transformation is correct. More formally:

Property 3.2 *Let P a process on A such that $clk \in A$ and $clk = tick(P)$.*
Let $(|P_1|P_2|)$ a subprocess of P (at a syntactical level), P_1 on A_1, P_2 on A_2, $A_1 \subseteq A$ and $A_2 \subseteq A$.
Then:

$$[\![\mathcal{I}(P_1|P_2)]\!] = [\![\mathcal{I}(P_1)|\mathcal{I}(P_2)]\!]$$

This property can be proved by equivalence of traces using some additional lemmas.

3.2.2 Producing HDL Code, the $\mathcal{H}$ Transformation

This transformation corresponds to the second step of the HDL generation process. Basically $\mathcal{H}$ applies a set of rules for a structural translation based on the chosen HDL target language. These rules translate operators, processes, and types of SIGNAL to the equivalent elements of the target HDL. For instance, when generating VHDL, the translation of the default operator is given figure 18. The mapping used for scalar types is the following: `event` is translated in BIT, `logical` is translated in BIT and `integer` is translated in INTEGER range MINVAL to MAXVAL.

Integers require a range to be provided for minimizing hardware. We envision a semi-automatic user-assisted process to extract such ranges. If a range is not known then default values have to be used depending on the intended internal/external bus widths.

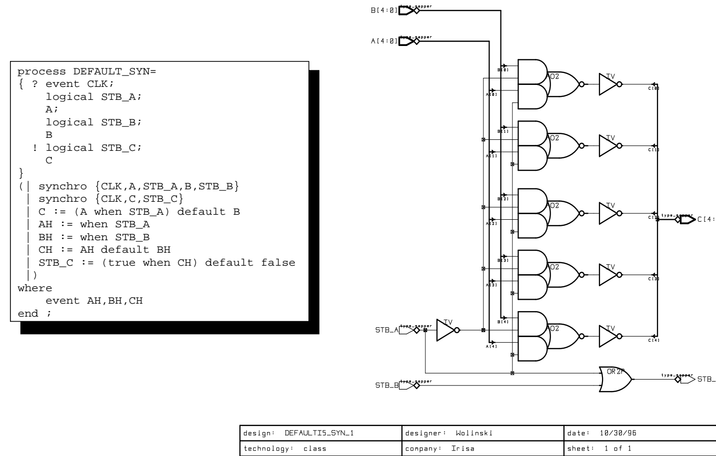


Figure 18: The default operator (left) and the synthesis result (right).

A prototype of the VHDL code generator has been implemented. The output of the VHDL synthesis process, for the incrementer-decrementer counter given in figure 11, is shown figure 19. The corresponding simulation is given figure 20.

Other circuits have been synthesized with our tool: an address generator (control protocol and data path), a GCD (based on euclid algorithm) and a eight words depth fifo. The results are summarized in Table 1. The areas are obtained using the *Class* library of the Synopsys [22] tool and after Synopsys optimization.

	SIGNAL	VHDL	gen. time	ports	nets	cells	Area		
							comb.	non comb.	total
Counter	17l	463l	0.24s	14	85	80	109	56	165
Address gen.	332l	1879l	0.31s	20	650	637	863	1024	1887
GCD	23l	607l	0.34s	29	175	155	213	112	325
FIFO	26l	763l	0.39s	22	646	634	798	952	1750

Table 1: Synthesis results

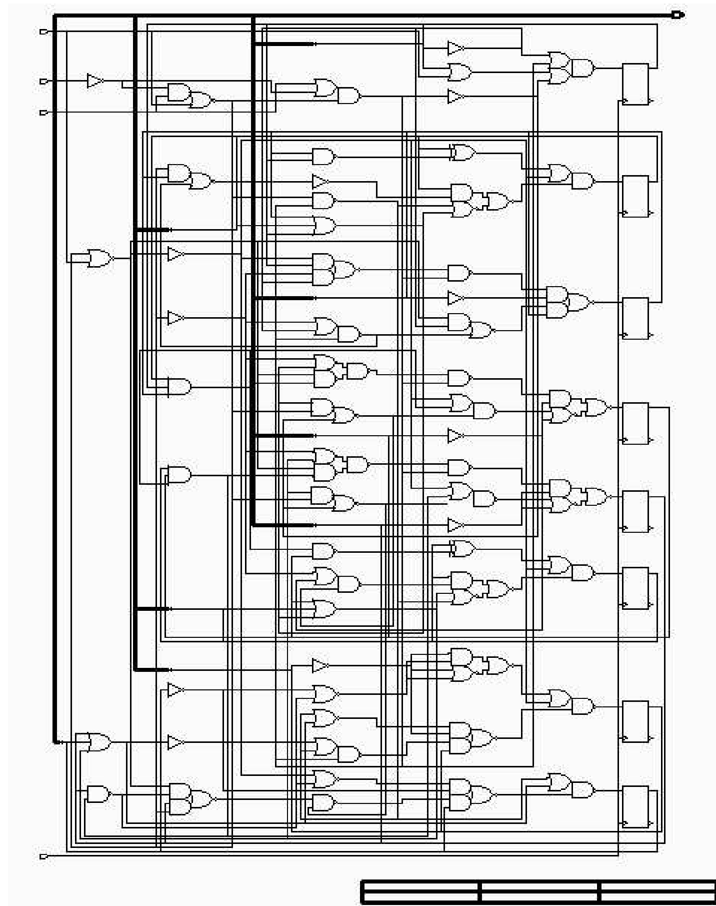


Figure 19: Synthesis of the Incr.-Decr. Counter.

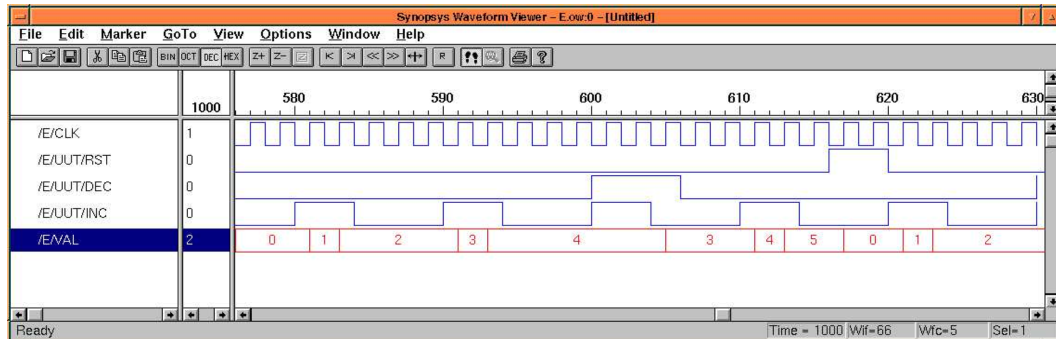


Figure 20: VHDL simulation of the Incr.-Decr. Counter.

4 Proving Properties on SIGNAL Specifications

To ensure that a SIGNAL specification is correct, simulation is insufficient. Verification of properties is necessary. The correctness properties that we can address are:

- Safety properties (nothing bad happens) like partial correctness, mutual exclusion or invariants. These statical properties have to be satisfied at all instants.

- Liveness properties (something good will happen) like accessibility or starvation freedom. These dynamical properties have to be satisfied on the flows of the SIGNAL program.

In the following of this section we present two methods of verification; the first one using observers and the SIGNAL compiler, the second one using a model checking tool, SIGALI that take into account SIGNAL $\perp$ event and the clock hierarchical structure.

4.1 Using the SIGNAL compiler

Part of the safety properties can be addressed directly by the SIGNAL compiler. The work done by the compiler consists in computing the control of the program. During this computation called “clock calculus”, the compiler

computes clocks of the signals, checks the consistency of constraints between these clocks, prove some statical properties and find inconsistencies like null clock expressions (signals without values) or data dependence cycles. This clock calculus relies on an algebra on set of instants. This is detailed in [1].

We can use this calculus to prove some statical properties. For instance if we want to prove that a SIGNAL specification S has the statical property P expressed in SIGNAL, we compile the following composition: $(| S \mid ERROR := not\ P |)$.

When the compiler computes a null clock for the signal $ERROR$ (which means that $ERROR$ is never true), the specification S satisfies the statical property P.

4.1.1 Example

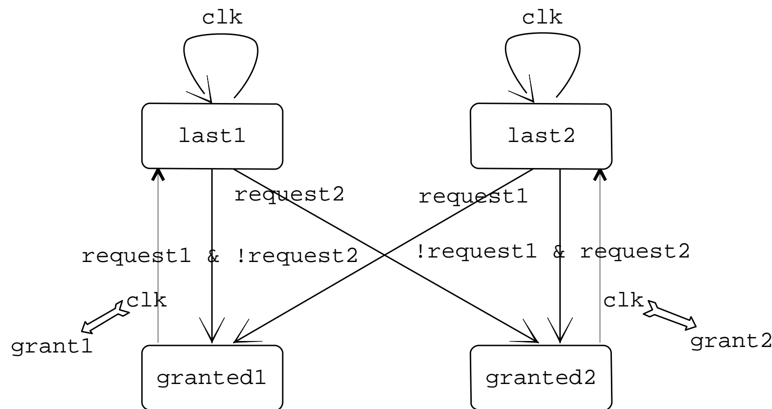


Figure 21: Control automaton of an arbiter.

Let us consider the specification of a bus arbiter given in graphical form figure 21. The behavior of this arbiter is the following:

- if one client requests the resource and the other does not request it, then the requesting client is granted.
- if the two clients request the resource at the same cycle then the client which has last been granted is not granted and the other is.

Using the method described above, we can prove that the two clients cannot be granted at the same cycle. In order to prove this safety property we compose the result of the signal generation from the automaton description and the expression:

$$ERROR := GRANT1 \ when \ GRANT2$$

After compilation, the signal $ERROR$ is a null clock signal and thus that $ERROR$ is never true and thus that two clients cannot be granted at the same cycle. However, this method faces the problem that when the compiler does not compute a null clock. In this case, there are two possibilities: the property is false or the compiler is not able to compute the null clock due to its heuristics.

4.2 Model Checking

We have seen that with the signal compiler, if a property is not true, it does not mean that it is false. The compiler only uses statical properties of SIGNAL and so cannot address dynamical properties of programs such as liveness properties. In such cases, a more exhaustive method such as model checking is needed.

We use the SIGALI model checker which is developed in the SIGNAL environment. The properties are written in the CTL temporal logic.

4.2.1 Overview of SIGALI

The equational nature of the SIGNAL language leads naturally to the use of methods based on system of polynomial dynamical equations over $\mathcal{GF}_3$ as a formal model of programs behavior [13]. The SIGALI model checker is based on this model.

The system of polynomial equations characterize sets of solutions which are states and events. The techniques used in SIGALI consist in manipuling the equation systems instead of the solution sets which avoids the enumeration of the state space. In order to allow the use of SIGALI, a SIGNAL process is translated into a system of polynomial equations over $\mathcal{GF}_3(\ true \rightarrow +1, \ false \rightarrow -1, \ absent \rightarrow 0, \ present \rightarrow \pm 1)$. An overview of this method for verifying SIGNAL programs is given in [14].

To limit cases exploration, SIGALI takes advantage of the partial order on signal's clock.

4.2.2 Overview of CTL (Computation tree logic)

This branching time temporal logic has first been proposed by Clarke and Emerson [6]. In their approach, a finite state system is modeled as a labeled state-transition graph which can be viewed as a finite Kripke structure [11]. In addition to usual temporal operators G (always), F (sometimes), X (next time) and U (until) which are regarded as state quantifiers, path quantifiers are provided to represent all paths (A) and some paths (E) from a given state. Some abbreviated operators are also provides: $AF\phi = A(true\ U\ \phi)$; $EF\phi = E(true\ U\ \phi)$; $AG\phi = \neg EF(\neg\phi)$ and $EG\phi = \neg AF(\neg\phi)$.

In order to be checked by SIGALI, CTL formulae are translated into expressions involving operations on polynomial dynamical systems.

4.2.3 Requirements

In order to be able to prove properties on SIGNAL specifications with model checking techniques, we need to have a boolean description. Particularly, the states of the automatons used to describe the control parts, that are translated into integers for the simulation step, have to be translated into boolean signals. In our prototype we use a binary coding of these integers.

4.2.4 Example

Let us consider again the arbiter given 21. If we consider the third transition, its translation with a boolean coding of the states is given in figure 22.

On this example, we can prove the same safety property as we did with the compiler. This property is expressed by the following CTL formula:

$$AG(\neg(GRANT1 \wedge GRANT2)) \quad (1)$$

As SIGALI works in $\mathcal{GF}_3$and thus considers three values for a boolean signal, we have, for instance, to make a difference between the $isfalse$ and the not operator. The operator $isfalse(b)$ tests if the value of b is -1 and $not(b)$ is the complementary set operator ($not(true) = \{false, absent\}$).

```
| stateCode_0_X_2_X := true when
   ( when ((not (stateCodeZ_0) and not (stateCodeZ_1))
   when (when (not (REQUEST1)) when (REQUEST2)) ) )
| stateCode_1_X_2_X := true when
   ( when ((not (stateCodeZ_0) and not (stateCodeZ_1))
   when (when (not (REQUEST1)) when (REQUEST2)) ) )
```

Figure 22: SIGNAL description of an arbiter with boolean states

So the formula (1) corresponds to the following SIGALI expression where *and* is the intersection set operator:

$$AG(not(and(istrue(GRANT1), istrue(GRANT2)))) \tag{2}$$

On this example, we can prove liveness properties too. For instance, we can prove that if a request is made by a client, it will eventually be granted. These property is expressed by the formula (3) which script of the corresponding SIGALI session is also given figure 23:

$$AG(imp(REQUEST1, AF(GRANT1))) \tag{3}$$

```
> read("ARBITER.z3z");
> read("Ctl.z3z");
> verif(AG(not(and(istrue(GRANT1_4),istrue(GRANT2_5)))));
True
> verif(AG(imp(REQUEST1_1,AF(GRANT1_4))));
True
```

Figure 23: A SIGALI proof session

5 Related Works

Many approaches to hardware specification relies on hardware description language such as Verilog or VHDL [23, 24]. As these languages are conceptually similar to general purpose imperative languages such as C, ADA or Fortran they tend to the same drawbacks:

1. HDL language descriptions are too close to implementations. The synchronous approach allows to write executable specifications which are more independent from any implementation.

2. In SIGNAL there is no notion of a synthesizable subsets of the language which are generally obstacles to ensure portability between tools.

3. Model checking techniques are difficult to implement on HDL (especially on the full language) while SIGNAL offers a formal model that already integrates clock calculus. Furthermore, it is possible to automatically transform programs into an equivalent form that improves accuracy of model checking. For instance, bounded integers can be transformed as a set of boolean variables for the purpose of model checking, while integers are used when generating the implementation or simulating it.

The methodology we are proposing can also be implemented on other synchronous languages than SIGNAL [3, 8]. However, SIGNAL offers a good compromise over other existing ones. Indeed, SIGNAL is a declarative language that naturally provides an abstraction of hardware components. The clock calculus of SIGNAL synthesizes the constraints and verifies their consistency (they admit a solution) and their completeness (they admit only one solution). For instance, compared to Statechart [10] we ensure the deterministic behavior of a component. For a more extensive comparison of synchronous languages the reader may refer to [9].

Hardware/software partitioning issues have not been considered so far, nevertheless our approach is also related to software/hardware co-design [20] environments for embedded system such as Polis [2], VULCAN [21], COSYMA [7], Chinook [5], etc. The closest to our approach is Polis. Others systems are either based on variations of the C language or on HDL language. Compared

to Polis, rather than having multiple front-ends to a finite state machine language, our approach is based on a unique formalism for all specification steps. However, a SIGNAL interface to Polis (i.e. a translator from SIGNAL to the Polis CFSM language) could be easily implemented to provide a link between the tools.

Conclusion

We present in this report a methodology based on the synchronous data flow SIGNAL language. We show that this language provides an adequate abstraction for the executable specification of complex hardware components. Generation of synthesizable HDL code can be done as well as model checking. This allows to integrate in a unique framework and many of the specification steps. However, this work still need to be extended to provide a very convenient tool. The first extension will be to enhance the SIGNAL language to allow more genericity in the description and also new data types. Part of future work will also consist in providing proven correct refinement transformations. None of these extensions are fundamental but they should make hardware specifications easier to write. The next direction of study is the integration of hardware components. In particular HDL to SIGNAL translation may be explored to use existing HDL libraries. Finally to improve the HDL code generation we will study optimizations based on the clock calculus, such as resource allocation to decrease hardware cost.

References

[1] T.A. Amabegnon, L. Besnard, and P. Le Guernic. Arborescent Canonical form of Boolean Expressions. Technical Report 2290, INRIA, June 1994.

[2] F. Balarin, M. Chiodo, D. Engels, P. Giusto, and etc. Polis, A design environment for control-dominated embedded systems, Usr's Manual. Technical report, University of California, etc., December 1996.

[3] G. Berry, P. Couronné, and G. Gonthier. Synchronous Programming of Reactive Systems, an Introduction to ESTEREL. In *K. Fuchi abd M.*

Nivat, editors, Programming of Future Generation Computers. Elsevier Science Publisher B.V. (North Holland), 1988.

[4] Patricia Bournai, Bruno Chéron, Thierry Gautier, Bernard Houssais, and Paul Le Guernic. Signal manual. Research Report 1969, INRIA, September 1993.

[5] Pai Chou, Ross Ortega, and Gaetano Borriello. The Chinook Hardware/Software Co-synthesis System. *Dept. of CS and Engineering, U. of Washington, Tech. Report 95-03-04*, March 1994.

[6] E. M. Clarke and Emerson A.A. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In Springer-Verlag, editor, *Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71, 1981.

[7] D.E.Thomas, J.K.Adams, and H.Schmitt. A Model and Methodology for Hardware-Software Codesign. *IEEE Design & Test of Computers*, pages 6–15, September 1993.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language LUSTRE. *Proceedings of the IEEE*, 79, September 1991.

[9] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[10] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8, 1987.

[11] G.E. Hughes and M.J. Creswell. *An Introduction to Modal Logic.* Methuen, 1977.

[12] Apostolos Kountouris and Paul Le Guernic. Profiling of signal programs and its application in the timing evaluation of design implementations. In *IEEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems*, HP Labs Bristol UK, February 1996.

[13] M. Le Borgne, A. Benveniste, and P. Le Guernic. Dynamical Systems over Galois Fields and DEDS Control Problems. In *33th IEEE Conf. on Decision and Control*, volume 3, pages 1505–1509, 1991.

[14] M. Le Borgne, H. Marchand, E. Rutten, and M. Samaan. Formal Verification of SIGNAL Programs: Application to a Power Transformer Station Controller. In Springer, editor, *5th Conf. on Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 271–285, July 1996.

[15] Paul Le Guernic and Albert Benveniste. Real-time synchronous, data-flow programming : The language SIGNAL and its mathematical semantics. Internal publication 298, IRISA, Rennes, June 1986.

[16] Paul Le Guernic and Thierry Gautier. Data-flow to von neumann : the signal approach. Rapport de Recherche 1229, INRIA, May 1990.

[17] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, septembre 1991.

[18] P. LeGuernic, T. Gautier abd M. LeBorgne, and C. LeMaire. Programming real time applications with signal. *Proceedings of the IEEE*, 79, September 1991.

[19] Esprit project EP 20897: Sacres. The Declarative Code DC+, Version 1.2. Technical report, SACRE CONSORTIUM, May 1996.

[20] R.K.Gupta and Giovanni De Micheli. Hardware-Software Cosynthesis for Digital Systems. *IEEE Design & Test of Computers*, pages 29–41, September 1993.

[21] Giovanni De Micheli R.K.Gupta. System Synthesis via Hardware-Software Codesign. *Computer Systems Laboratory Technical Report CSL-TR-92-548*, 1992.

[22] Inc. Synopsys, editor. *HDL Compiler for Verilog Reference Manual, Version 3.5.* February 1996.

[23] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1994.

[24] *IEEE standard VHDL: language reference manual*. IEEE, 1987. std 1076-1987.