



# Toward Static Analysis of SIGNAL Programs using Interval Techniques

Abdoulaye Gamatié, Thierry Gautier, Paul Le Guernic

## ► To cite this version:

Abdoulaye Gamatié, Thierry Gautier, Paul Le Guernic. Toward Static Analysis of SIGNAL Programs using Interval Techniques. Synchronous Languages, Applications, and Programming (SLAP 2006), Mar 2006, Vienna, Austria. pp.SLAP 2006. hal-00544123

**HAL Id: hal-00544123**

**<https://hal.science/hal-00544123>**

Submitted on 7 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.

# Towards Static Analysis of SIGNAL Programs using Interval Techniques

Abdoulaye GAMATIÉ<sup>1</sup>,

*INRIA - Synergie Park, rue Pierre et Marie Curie 59062 - Lezennes, France*

Thierry GAUTIER, Paul LE GUERNIC<sup>2,3</sup>

*IRISA/INRIA - Campus de Beaulieu - 35042 Rennes cedex, France*

---

## Abstract

This paper presents a work-in-progress aiming at improving the functional analysis of SIGNAL programs. The usual adopted technique relies on abstractions. Typically, in order to check the presence or absence of variables in a program at some logical instants, the program is transformed into another program that reflects its clock information so that the presence or absence of each variable can be straightforwardly checked. SIGNAL adopts a boolean abstraction for the static functional analysis of programs. This abstraction does not enable to fully reason on the values of non logical variables. Here, we propose a solution based on interval techniques in order to be able to deal with both logical and numerical parts of programs.

---

## 1 Motivations

Safety-critical systems (e.g. medical, automotive, avionics systems) are increasingly evolving in complexity and functionalities. Their design consequently requires reliable technologies allowing to unambiguously describe and analyze their behaviors so as to meet their stringent requirements. Over the last decade, the synchronous approach [4] has demonstrated its efficiency to cope with design problems inherent to such systems. In particular, verification issues are addressed using trustworthy formal techniques.

*Model-checking* [9] is part of these techniques. It associates a system with a finite discrete model (e.g. a finite state machine) against which properties are checked (e.g. reachability, liveness). Both model and properties can be

---

<sup>1</sup> abdoulaye.gamatie@lifl.fr

<sup>2</sup> thierry.gautier@irisa.fr

<sup>3</sup> paul.leguernic@irisa.fr

specified with formalisms such as the synchronous languages. The `SIGALI` tool [19] associated with the synchronous language `SIGNAL` adopts this technique. While model-checking has been successfully used in several studies, it sometimes appears impractical. Typically, when a finite model does not exist for a system or when the properties of interest involve non-linear numerical terms. A verification technique that suits more in these cases is *theorem proving* [12]. It allows one to prove some properties from a set of “facts” (axioms of the system) and deduction rules according to the underlying logic. Examples of theorem provers are `Coq` [24] and `Pvs` [21]. The semantics of `SIGNAL` has been formalized in `Coq`. A few studies based on this formalization addressed the correctness issues of a steam-boiler [15] and a protocol for loosely time-triggered architectures [16]. Although theorem proving is a powerful technique, a main drawback is that the associated tools often require interaction with users. The verification process therefore becomes slow, even error-prone. The last technique we mention is *abstract interpretation* [11], which is a theory of discrete approximation of the semantics of languages. The central idea is that the precision of the semantics depends on the considered level of observation. The coarser is the approximation, the less precise but computable version is yielded by this approximation. While all questions cannot be addressed, those answered through the effective computation of the approximate semantics are always correct. Abstract interpretation is mainly applied to static analysis.

In this study, we use abstract interpretation to statically address the functional properties of `SIGNAL` programs. The analysis process currently relies on an abstraction on the boolean domain. While the values of boolean variables are fully taken into account, it is not the case for numerical variables. More generally, one can observe that while synchronous design environments provide several verification tools, only a few of them enable to address numerical properties. Here, we combine interval techniques with boolean calculus in order to address both logical and numerical properties of `SIGNAL` programs. Intervals are very interesting abstract domains for several reasons. First, they enable approximations that can be easily manipulated. They are powerful enough to be considered in abstract interpretation [10] or in numerical analysis [1]. Second, interval techniques have been successfully experimented for an efficient manipulation of boolean formulas. The result is the definition of *interval decision diagrams* (IDDs) [23], which are considered in this paper.

In the next, Section 2 briefly introduces the abstract interpretation notions. Then, Section 3 presents the `SIGNAL` language (semantics and analysis). Section 4 defines an approximation of `SIGNAL` programs using IDDs in order to address numerical properties. Finally, Section 5 gives conclusions and perspectives.

## 2 Abstract interpretation

Introduced by Cousot [11], abstract interpretation is a general theory for discrete approximation of the semantics of systems based on monotonic functions

over ordered sets. The static analysis is stated as a formal correspondence between the concrete semantics of a program and its abstract semantics with respect to the property to be verified. Two basic transformations are distinguished: an *abstraction function*  $\alpha : \mathcal{C} \rightarrow \mathcal{A}$  and a *concretization function*  $\gamma : \mathcal{A} \rightarrow \mathcal{C}$ , where  $\mathcal{C}$  and  $\mathcal{A}$  respectively denote the concrete and abstract domains. The function  $\alpha$  associates each concrete element with an abstract element that reflects its properties whereas  $\gamma$  defines the sub-set of concrete elements that satisfy the property characterized by an abstract element.

To define a correct abstract interpretation, the tuple  $(\mathcal{C}, \mathcal{A}, \alpha, \gamma)$  must satisfy the following properties:

- $\mathcal{C}$  is a *complete partially ordered set*, meaning that it admits an *inf*  $\perp_{\mathcal{C}}$  element and a partial order  $\preceq$  relation such that:  $\forall c \in \mathcal{C}, \perp_{\mathcal{C}} \preceq c$ .  $\mathcal{A}$  is a *complete lattice*, i.e. a partially ordered set with *inf*  $\perp_{\mathcal{A}}$  and *sup*  $\top$  elements, and *meet*  $\sqcap$  and *join*  $\sqcup$  operators, which admits a partial order  $\sqsubseteq$  relation such that:  $\forall a \in \mathcal{A}, \perp_{\mathcal{A}} \sqsubseteq a, a \sqsubseteq \top_{\mathcal{A}}$ .
- The abstraction function  $\alpha$  is first extended to the domain  $\mathcal{P}(\mathcal{C})$  as follows:  $\alpha : \mathcal{P}(\mathcal{C}) \rightarrow \mathcal{A}$ , where  $\forall C \subseteq \mathcal{C} \alpha(C) = \bigcup_{c \in C} \alpha(c)$ . Then, the following correctness criteria must be verified:

$$\begin{cases} Id \subseteq \gamma \circ \alpha & \Leftrightarrow \quad \forall c \in \mathcal{C} \ c \in \gamma(\alpha(c)), \\ Id = \alpha \circ \gamma & \Leftrightarrow \quad \forall a \in \mathcal{A} \ a = \alpha(\gamma(a)). \end{cases}$$

Furthermore, if  $f$  is a function defined on  $\mathcal{C}$  and  $g$  is an associated abstract version defined on  $\mathcal{A}$ , then  $g$  is a correct approximation of  $f$  if  $f \subseteq \gamma \circ g \circ \alpha$  or equivalently  $\forall c \in \mathcal{C}, f(c) \in \gamma(g(\alpha(c)))$ .

The above notions are used in Section 4.2 to define an approximation of SIGNAL programs using intervals as abstract domain for numerical variables.

### 3 The SIGNAL language

SIGNAL [17] handles unbounded series of typed values, implicitly indexed by discrete time and called *signals*. At a given logical instant, a signal may be present or absent (denoted by  $\perp$ ). There is a particular type of signal called **event**, which is always *true* when it is present. The set of instants where a signal  $x$  is present is referred to as its *clock*, noted as  $\hat{x}$  in the language. Signals that have the same clock are said to be *synchronous*. A *process* is a system of equations over signals. A program is a process. In the sequel, we first introduce the denotational semantics of SIGNAL, which will serve in the definition of the abstract interpretation in Section 4.2. Then, we present the functional analysis of programs based on syntactic abstractions (Section 3.2).

#### 3.1 A denotational semantics

We present the basic notions of a denotational semantics (also called *trace semantics*), which is used to give the semantics of SIGNAL primitive constructs.

### 3.1.1 Basic notions

Let us consider a finite set  $X = \{x_1, \dots, x_n\}$  of typed variables called *ports*. For each  $x_i \in X$ ,  $\mathcal{D}_{x_i}$  is the domain of values (e.g. integer, real, boolean) that may be held by  $x_i$  at every instant. In addition, we have:

$$\mathcal{D} = \bigcup_{i=1}^n \mathcal{D}_{x_i} \text{ and } \mathcal{D}^\perp = \mathcal{D} \cup \{\perp\},$$

where  $\perp \notin \mathcal{D}$  denotes the absence of value associated with a port at a given instant.  $\mathcal{D}_{x_i}^\perp$  and  $\mathcal{D}_{X1}^\perp$  are defined in the same way ( $X1$  is a subset of  $X$ ).

For every non-empty subset  $X1$  of  $X$ , we consider the following definitions:

- The set of applications  $m$  defined from  $X1$  to  $\mathcal{D}_{X1}^\perp$ , called set of *events* on  $X1$ , is denoted by  $\mathcal{E}_{X1}$ . The expression  $m(x) = \perp$  means that  $x$  does not hold a value;  $m(x) = v$  means that  $x$  holds the value  $v$ ; and  $m(X1) = \{y | x \in X1, m(x) = y\}$ . The set of events on  $X1$  is denoted by  $\mathcal{E}_{X1} = X1 \rightarrow \mathcal{D}_{X1}^\perp$ , and the set of all possible events is therefore  $\mathcal{E} = \bigcup_{X1 \subseteq X} \mathcal{E}_{X1}$ . The event on an empty set of ports is represented by  $\mathcal{E}_\emptyset = \{\emptyset\}$ .
- The set of applications  $T$  defined from the set  $\mathbb{N}$  of natural integers to  $\mathcal{E}_{X1}$ , called set of *traces* on  $X1$ , is denoted by  $\mathcal{T}_{X1}^\perp : \mathbb{N} \rightarrow \mathcal{E}_{X1}$ . The set of all possible traces is therefore  $\mathcal{T}^\perp = \bigcup_{X1 \subseteq X} \mathcal{T}_{X1}^\perp$ . Moreover, we have  $\mathcal{T}_\emptyset = \mathbf{1} = \mathbb{N} \rightarrow \mathcal{E}_\emptyset$  and  $\mathbf{0} = \mathbb{N} \rightarrow (X \rightarrow \{\perp\})$ .
- For  $X2 \subset X1$  and  $T$  being defined on  $X1$ , the *restriction* of  $T(t)$  to  $X2$ , noted  $X2.T : \mathbb{N} \rightarrow \mathcal{E}_{X2}$ , satisfies:  $\forall t \in \mathbb{N}, \forall x \in X2 \quad X2.T(t)(x) = T(t)(x)$ . We have  $\emptyset.T \in \mathcal{T}_\emptyset$  (which is a singleton).
- A *stream* on  $X1 \subseteq X$  represents any trace  $T$  of  $\mathcal{T}_{X1}^\perp$  such that:

$$\exists t (T(t)(X1) = \{\perp\}) \Rightarrow \forall s \geq t \quad T(s)(X1) = \{\perp\}.$$

We denote by  $\mathcal{T}$  (resp.  $\mathcal{T}_{X1}$ ) the set of streams of  $\mathcal{T}^\perp$  (resp.  $\mathcal{T}_{X1}^\perp$ ).  $0_{\mathcal{T}}$  is defined on  $\mathbb{N} \rightarrow (X \rightarrow \{\perp\})$  and  $\emptyset.T$  is a stream.

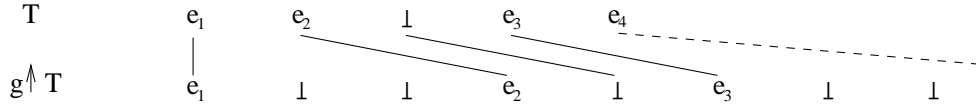


Figure 1. Temporal expansion.

- Let  $g$  be a strictly increasing application  $\mathbb{N} \rightarrow \mathbb{N}$  and  $T : \mathbb{N} \rightarrow X1 \rightarrow \mathcal{D}_{X1}^\perp$ ,  $X1 \neq \emptyset$ , a trace on  $X1$ . The *expansion* of  $T$  by  $g$  (i.e. insertion of  $\perp$ , see FIG. 1) is the trace  $g \uparrow T : \mathbb{N} \rightarrow X1 \rightarrow \mathcal{D}_{X1}^\perp$  defined as  $(g \uparrow T) \circ g = T$  where:  $\forall t \forall s \quad g(t) < s < g(t+1) \quad (g \uparrow T)(s)(X1) = \{\perp\}$ . Here,  $g$  is referred to as expansion function. Finally, the following holds:  $g \uparrow T_\emptyset = T_\emptyset$ .

### 3.1.2 Semantic definition of the primitive constructs

The whole SIGNAL language relies on six primitive constructs (see below). The semantics associated with each construct is defined in terms of set of streams.

Formally, a process on  $X1 \subseteq X$  is a set of constrained streams on  $X1$  (i.e., a subset of  $\mathcal{T}_{X1}$ ). Each SIGNAL statement defining a process  $P$  is associated with the process  $\llbracket P \rrbracket$  that denotes its semantics.

*Functions/Relations.*  $\mathbf{x}_{n+1} := \mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n)$ .

$$\begin{aligned} \llbracket P \rrbracket &= \{ T \in \mathcal{T}_{\{x_1, \dots, x_n, x_{n+1}\}} / \forall t \\ &\quad T(t)(\{x_1, \dots, x_{n+1}\}) = \{\perp\} \text{ or } v_{n+1} = f(v_1, \dots, v_n), v_i = T(t)(x_i) \} \end{aligned}$$

*Delay.*  $\mathbf{x}_2 := \mathbf{x}_1 \text{ \$1 init } v_0$

$$\begin{aligned} \llbracket P \rrbracket &= \{ T \in \mathcal{T}_{\{x_1, x_2\}} / \forall t \\ &\quad (T(t)(\{x_1, x_2\}) = \{\perp\}) \text{ or } (t > 0 \text{ and } T(t)(x_2) = T(t-1)(x_1)) \\ &\quad \text{or } (t = 0 \text{ and } T(0)(x_2) = v_0 \text{ and } T(0)(x_1) \neq \perp) \} \end{aligned}$$

*Under-sampling.*  $\mathbf{x}_3 := \mathbf{x}_1 \text{ when } \mathbf{x}_2$

$$\begin{aligned} \llbracket P \rrbracket &= \{ T \in \mathcal{T}_{\{x_1, x_2, x_3\}} / \forall t \\ &\quad T(t)(x_2) = \text{true} \Rightarrow T(t)(x_3) = T(t)(x_1) \text{ or } T(t)(x_2) \neq \text{true} \Rightarrow T(t)(x_3) = \perp \} \end{aligned}$$

*Deterministic merging.*  $\mathbf{x}_3 := \mathbf{x}_1 \text{ default } \mathbf{x}_2$

$$\begin{aligned} \llbracket P \rrbracket &= \{ T \in \mathcal{T}_{\{x_1, x_2, x_3\}} / \forall t \\ &\quad T(t)(x_1) \neq \perp \Rightarrow T(t)(x_3) = T(t)(x_1) \text{ or } T(t)(x_1) = \perp \Rightarrow T(t)(x_3) = T(t)(x_2) \} \end{aligned}$$

*Synchronous composition.*  $P1 \mid P2$  such that  $\llbracket P1 \rrbracket \subseteq \mathcal{T}_{X1}, \llbracket P2 \rrbracket \subseteq \mathcal{T}_{X2}$

$$\begin{aligned} \llbracket P \rrbracket &= \{ T \in \mathcal{T}_{X1 \cup X2} / \\ &\quad \exists T1 \in \llbracket P1 \rrbracket, \exists T2 \in \llbracket P2 \rrbracket, \exists f1, f2 \text{ expansion functions such that} \\ &\quad X1.T = f1 \upharpoonright T1, X2.T = f2 \upharpoonright T2 \} \end{aligned}$$

*Hiding.*  $P1 \text{ where } \mathbf{x}_1, \dots, \mathbf{x}_n$  such that  $\llbracket P1 \rrbracket \subseteq \mathcal{T}_{X1}$

$$\llbracket P \rrbracket = \{ T \in \mathcal{T}_{X1 - \{x_1, \dots, x_n\}} / \exists T1 \in \llbracket P1 \rrbracket, X1 - \{x_1, \dots, x_n\}.T1 = T \}$$

The same symbol is used to denote the composition in the syntactic and semantic domains, then:  $\llbracket P \rrbracket \mid \llbracket P' \rrbracket = \llbracket P \mid P' \rrbracket$ . The smallest set of variables associated with  $P$  is denoted by  $\text{vars}(P)$ . Finally, the projection on a set  $V$  of variables (which corresponds to applying the *hiding* operator on the complementary of  $V$  in  $\text{vars}(P)$ ) is denoted as  $\llbracket P \rrbracket|_V$ .

### 3.2 Functional analysis of programs

The functional properties of a program  $P$  include *invariant properties* as well as *dynamic properties*. The SIGNAL compiler itself addresses only invariant properties. A major part of its task is referred to as the *clock calculus*. Dynamic properties (e.g. reachability of some state, liveness), which are not in the scope of this paper, are addressed using SIGNAL [19]. The verification of functional properties of  $P$  relies on abstractions of  $P$ , which suit for the target analysis. These abstractions are defined using syntactic transformations.

### 3.2.1 Syntactic abstraction

A SIGNAL program  $P$  specifies on the one hand relations between clocks of signals, and on the other hand values of signals. Specific properties of  $P$  can be addressed by separating both aspects. In particular,  $P$  can be *abstracted* by a new program  $P'$ , which only describes its clock information. For instance, to check the absence of reaction in  $P$ , one just needs to focus on  $P'$ .

A possible way to derive  $P'$  from  $P$  is to use syntactic transformations. Basically, from the semantics of the composition operator, we can state that for any  $P, P'$ :  $(\llbracket P \rrbracket \mid \llbracket P' \rrbracket)_{\text{vars}(P)} \subseteq \llbracket P \rrbracket$ . More precisely, the following can be proved:  $\llbracket P \rrbracket = (\llbracket P \rrbracket \mid \llbracket P' \rrbracket)_{\text{vars}(P)}$  iff  $\llbracket P \rrbracket_{\text{vars}(P')} \subseteq \llbracket P' \rrbracket_{\text{vars}(P)}$ . In other words, every stream that belongs to  $\llbracket P \rrbracket$  is consistent with a stream from  $\llbracket P' \rrbracket$ . Hence, if  $\text{vars}(P) = \text{vars}(P')$  then:  $\llbracket P \rrbracket \mid \llbracket P' \rrbracket = \llbracket P \rrbracket$  iff  $\llbracket P \rrbracket \subseteq \llbracket P' \rrbracket$ . If  $P'$  denotes some property,  $P$  satisfies  $P'$  iff the previous equivalence holds.

A *syntactic abstraction* of  $P$  is based on a decomposition of  $P$  that syntactically derives from  $P$  another program  $\alpha(P)$  that abstracts  $P$ . It often leads to a pairwise decomposition of  $P$ , e.g.: *dynamic* part (induced by the *delay* operator) *vs* *static* part (induced by the other primitive constructs); *control* part (boolean and **event** signals) *vs* *computation* part (signals of other types).

**Definition 3.1** [syntactic abstraction] Given a process  $P$ , the process  $\alpha(P)$  such that  $\text{vars}(\alpha(P)) \subseteq \text{vars}(P)$  results from a syntactic abstraction  $\alpha$  by decomposition of  $P$  iff:  $\llbracket P \rrbracket \mid \llbracket \alpha(P) \rrbracket = \llbracket P \rrbracket$  or equivalently  $\llbracket P \rrbracket \subseteq \llbracket \alpha(P) \rrbracket$ .

We can notice a *complementarity* relation between  $\alpha(P)$  and a subpart of  $P$ . If  $\rho(P)$  denotes such a subpart ( $\rho$  stands for “residual”), each abstraction of  $P$  can be associated with a syntactic decomposition  $(\alpha, \rho)$  given by a set of rewriting rules of the form:

$$P \longrightarrow \alpha(P) \mid \rho(P) \text{ satisfying } \llbracket P \rrbracket = \llbracket \alpha(P) \rrbracket \mid \llbracket \rho(P) \rrbracket.$$

One can prove that  $\alpha(P)$  is actually an abstraction of  $P$  [20]. Moreover, the abstraction  $\alpha$  resulting from this decomposition must be idempotent:  $\llbracket \alpha(\alpha(P)) \rrbracket = \llbracket \alpha(P) \rrbracket$ .

**Definition 3.2** [syntactic decomposition] A syntactic decomposition is a pair of transformations  $(\alpha, \rho)$  s.t. for all primitive constructs  $P$ :  $\llbracket \alpha(P) \rrbracket \mid \llbracket \rho(P) \rrbracket = \llbracket P \rrbracket$  and  $\alpha$  is idempotent. Thus, the syntactic decomposition  $(\alpha, \rho)$  is defined inductively for composition and restriction:

$$\begin{aligned} \alpha(P_1 \mid P_2) &= \alpha(P_1) \mid \alpha(P_2) \quad \text{and} \quad \rho(P_1 \mid P_2) = \rho(P_1) \mid \rho(P_2) \\ \alpha(P \text{ where } x) &= \alpha(P) \text{ where } x \quad \text{and} \quad \rho(P \text{ where } x) = \rho(P) \text{ where } x \end{aligned}$$

### 3.2.2 Application: control abstraction

The control part of a SIGNAL program  $P$  consists of the synchronization and boolean signals of  $P$ . The abstraction of the boolean part of  $P$  is obtained by extracting from  $P$  the definition of boolean variables whereas the synchronization abstraction is given by the clock relations specified in  $P$ . We concentrate on



the synchronization sub-part in order to show the limits of the static analysis adopted in the SIGNAL compiler, and then motivate our idea.

The abstraction of the synchronization sub-part of  $P$  is a program  $\alpha_{sync}(P)$  that corresponds to the largest relation between the clocks of signals of  $P$ . In SIGNAL, clocks and their associated relations are formalized through a *clock algebra* [2], which we denote by  $\mathcal{C}$  s.t.:  $\mathcal{C} = \langle I, \cap, \cup, \setminus, \emptyset \rangle$ , where  $I$  and  $\emptyset$  respectively denote a reference set of instants and the empty clock. Given a set of *clock variables* interpreted as subsets of  $I$  (containing for instance  $\hat{x}$ ),  $\mathcal{C}$  can represent relations like  $\hat{x} = \hat{y} \cup \hat{z}$ . The set inclusion  $\subseteq$  operator expresses that a clock is a subset of another clock. FIG. 2 illustrates  $\alpha_{sync}$  and its associated formalization in  $\mathcal{C}$ .

$P$	$\alpha_{sync}(P)$	constraints in $\mathcal{C}$
$x_{n+1} := f(x_1, \dots, x_n)$	$x_{n+1} \hat{=} x_1 \hat{=} \dots \hat{=} x_n$	$\hat{x}_{n+1} = \hat{x}_1 = \dots = \hat{x}_n$
$x_2 := x_1 \text{ \$1 init } v_0$	$x_2 \hat{=} x_1$	$\hat{x}_2 = \hat{x}_1$
$x_3 := x_1 \text{ when } x_2$	$x_3 \hat{=} x_1 \text{ when } x_2$	$\hat{x}_3 = \hat{x}_1 \cap [x_2], \begin{cases} [x_2] \cup [\neg x_2] = \hat{x}_2, \\ [x_2] \cap [\neg x_2] = \emptyset \end{cases}$
$x_3 := x_1 \text{ default } x_2$	$x_3 \hat{=} x_1 \hat{+} x_2$	$\hat{x}_3 = \hat{x}_1 \cup \hat{x}_2$

Figure 2. Synchronizations in SIGNAL.

There is a complementarity relation between  $\alpha_{sync}(P)$  and  $P$  (i.e.  $\rho_{sync}(P) = P$ ). Since  $\alpha_{sync}(P)$  does not induce any supplementary constraint on  $P$  when composed, we trivially prove that:  $\llbracket \alpha_{sync}(P) \rrbracket \mid \llbracket P \rrbracket = \llbracket P \rrbracket$ .

In the clock algebra, a *condition-clock* denoted by  $[x_2]$  is introduced for the *under-sampling* operator. It denotes the set of instants where the boolean expression  $x_2$  is present and *true* ( $[\neg x_2]$  corresponds to *false*). Note the partitioning of  $\hat{x}_2$ , which is defined using  $[x_2]$  and  $[\neg x_2]$ . When  $x_2$  is defined by some numerical operation (typically a comparison),  $[x_2]$  and  $[\neg x_2]$  are seen as “black boxes” that abstract, together with the partitioning, the value of  $x_2$ .

**Example 3.3** Let us consider the following SIGNAL program:

```

(| ...
  | x1 := f(N1 when x > 2*N)           (s1)
  | x2 := g(N2 when x < N)             (s2)
|)

```

where signals  $x$ ,  $x_1$ ,  $x_2$  and constants  $N$ ,  $N_1$ ,  $N_2$  are of integer type;  $f$  and  $g$  represent two numerical functions. The corresponding abstraction in the clock algebra is as follows:

$$(s1) \Rightarrow \hat{x}_1 = [c_1], \quad [c_1] \cup [\neg c_1] = \hat{c}_1, \quad [c_1] \cap [\neg c_1] = \emptyset \quad (c_1 \equiv x > 2N),$$

$$(s2) \Rightarrow \hat{x}_2 = [c_2], \quad [c_2] \cup [\neg c_2] = \hat{c}_2, \quad [c_2] \cap [\neg c_2] = \emptyset \quad (c_2 \equiv x < N)$$

Now, consider that a new equation defined by a function with  $x_1$  and  $x_2$  as arguments is added to the same program. The following synchronization is therefore induced on  $x_1$  and  $x_2$ :  $\hat{x}_1 = \hat{x}_2$ . It appears some clock inconsistency



since the instants at which these signals are defined are exclusive:  $[c_1] \cap [c_2] = \emptyset$ . However, the clock constraint representing this inconsistency between  $[c_1]$  and  $[c_2]$  does not belong to the set of constraints derived from the program following the above abstraction rules. As a matter of fact, expressions  $c_1$  and  $c_2$  are not interpreted since  $[c_1]$  and  $[c_2]$  are considered as black boxes. So, the compilation process cannot fix such a clock inconsistency.

### 3.2.3 Boolean abstraction of the control

In [2], a correspondence between clock algebra and boolean functions is defined. If one considers the propositional calculus, the following straightforward encoding is therefore obtained [20]. First, each clock variable  $\hat{x}$  is associated with a propositional variable  $b_x$ . Condition-clocks such as  $[c]$  are also encoded into  $b_{[c]}$ . Then, the operators on sets considered in  $\mathcal{C}$  are associated with the suitable characteristic functions. An informal example is as follows:

Clock algebra	$I$	$\mathbb{O}$	$\hat{x}_1 \cap \hat{x}_2 = \hat{x}_3 \cup \hat{x}_4$	$\hat{x}_1 \setminus \hat{x}_2 = \hat{x}_3$
Prop. calculus	<i>true</i>	<i>false</i>	$b_{x_1} \wedge b_{x_2} \Leftrightarrow b_{x_3} \vee b_{x_4}$	$b_{x_1} \wedge \neg b_{x_2} \Leftrightarrow b_{x_3}$

This encoding allows the executable code to handle clock variables as propositional ones and not as sets of instants. The advantage is that BDD [6] packages can be used to efficiently implement clock manipulation in the compiler. Note that in the LUSTRE language, clocks are also represented by special boolean variables. From now, clocks and their associated operations may be considered as propositional variables and boolean operators.

The control abstraction introduced in this section enables to fully deal with values of boolean signals. However, it is not the case for numerical signals. In particular, numerical expressions specified with condition-clocks (such as  $c_1$  and  $c_2$  in the above example), are not fully addressed. The consequence is that they cannot be compared so as to detect that they are exclusive. This reduces the power of the static analysis. The next section proposes an interval-based abstraction that allows to overcome this limitation.

## 4 Static analysis of SIGNAL programs using intervals

A few investigations have been already done in order to cope with numerical properties of SIGNAL programs [5] [20]. While these studies showed promising theoretical results, they did not unfortunately lead to the implementation of the results. One reason is the complexity of the proposed solutions. In our proposition, implementation issues are among major concerns. This proposition aims to extend the clock calculus of SIGNAL in order to be able to reason on both numerical and logical subparts of programs. We consider intervals as abstract domains for numerical variables. As illustrated in [10], intervals favor a simple and efficient way to compute approximations for numerical values.

We introduce *interval decision diagrams* (IDDs) that consist of directed

acyclic graph structures where each non terminal node corresponds to a test on an integer variable, and terminal nodes are propositional formula represented by BDDs. Each outgoing edge from a non terminal node is associated with an interval within the domain of the variable attached to the node. Each edge is linked either to another non terminal node or to a terminal node.

#### 4.1 Interval Decision Diagrams

The definition of IDDs given here slightly differs from the classical one [23] [8] where terminal nodes are mainly boolean literals *true* or *false* instead of complex propositional formulas. So, we consider a more general definition. Let us assume the following sets:

- $\mathbb{D}$  is a totally ordered set of numeric constant values.
- $\mathbb{X} = \{x_1, \dots, x_k\}$  is a finite set of variables defined on  $\mathbb{D}$ .
- $\mathbb{B} = \{b_1, \dots, b'_k\}$  is a finite set of boolean variables where  $\mathbb{X} \cap \mathbb{B} = \emptyset$ . We represent by  $\mathcal{B}$  the set of all boolean formulas on  $\mathbb{B}$ .

**Definition 4.1** Let  $x$  be an integer variable defined on  $D \subseteq \mathbb{D}$  and  $t$  a predicate logic formula on  $X \subseteq \mathbb{X}$ .  $t$  is an IDD node iff one of the following holds:

- $t \in \mathcal{B} \cup \{true, false\}$ ,
- $t = (x \in I_0 \wedge t_0) \vee (x \in I_1 \wedge t_1) \vee \dots \vee (x \in I_k \wedge t_k)$ ,

where  $(I_i)_{i \leq k}$  is a partition of  $D$  and  $(t_i)_{i \leq k}$  a set of IDD nodes. The node  $t$  is a predecessor of each  $(t_i)_{i \leq k}$ .

An IDD root does not have any predecessor. A set of IDD nodes  $(t_i)_{i < n}$  is said to be consistent if there is a unique root. Moreover, if  $t$  is an IDD node, let  $var(t)$  be the function that gives the non boolean variable tested on  $t$ :

$$var(t) = \begin{cases} x, & \text{if } t = (x \in I_0 \wedge t_0) \vee (x \in I_1 \wedge t_1) \vee \dots \vee (x \in I_k \wedge t_k) \\ t & \text{if } t \in \mathcal{B} \end{cases}$$

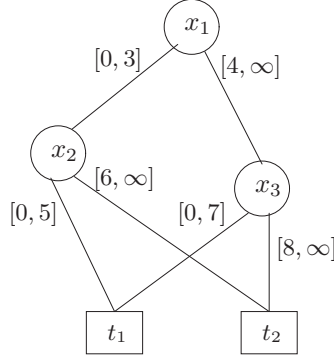
On the other hand,  $I = ((t_i)_{i < n}, \succ)$  is an IDD iff  $(t_i)_{i < n}$  is a consistent set of IDD nodes and  $\succ$  is an order on  $\mathbb{X} \cup \mathcal{B}$  such that  $\forall t \in (t_i)_{i < n}$  verifying  $t = (x \in I_0 \wedge t'_0) \vee (x \in I_1 \wedge t'_1) \vee \dots \vee (x \in I_k \wedge t'_k)$ , we have:

$$(x \succ var(t'_i) \text{ for each } i \leq k) \text{ and } (x \succ t' \text{ for each } t' \in \mathcal{B} \cup \{true, false\}).$$

FIG. 3 depicts an IDD associated with the following predicate:

$$p(x_1, x_2, x_3) = \begin{cases} t_1 & \text{if } (x_1 \in [0, 3] \wedge x_2 \in [0, 5]) \vee (x_1 \in [4, +\infty[ \wedge x_3 \in [0, 7]) \\ t_2 & \text{if } (x_1 \in [0, 3] \wedge x_2 \in [6, +\infty[) \vee (x_1 \in [4, +\infty[ \wedge x_3 \in [8, +\infty[). \end{cases}$$

Non-terminal nodes (circles) consist of numerical variables (e.g. the root  $x_1$ ) whereas the leaves (squares) are boolean formulas represented by BDDs (e.g.  $t_1$  and  $t_2$ ). Every outgoing edge associated with a non-terminal node is labelled with a possible range value of that node (i.e. an interval). As for logic formulas, we can perform all the usual logical operations on IDDs like negation ( $\neg$ ),

Figure 3. Interval decision diagram associated with  $p(x_1, x_2, x_3)$ .

conjunction ( $\wedge$ ), disjunction ( $\vee$ ), equivalence ( $\Leftrightarrow$ ), etc. An operational implementation of IDDs already exists [7]. However, this version must be slightly modified in order to take into account the more general definition considered here. A solution consists in extending the existing IDDs implementation with a BDD package implementation (e.g. the Buddy package [18]). Both implementations propose optimizations that are performed on the corresponding structures in order to prune redundant nodes and subtrees.

#### 4.2 Abstraction of programs using intervals

In order to approximate SIGNAL programs, we consider the following steps:

- (i) Abstraction of SIGNAL primitives using intervals,
- (ii) Translation of the resulting descriptions into IDDs.

Every variable  $\mathbf{x}$  of a program  $P$  is associated with an abstract numerical variable  $i_{\mathbf{x}}$  in the program resulting from the abstraction of  $P$ . The variable  $i_{\mathbf{x}}$  takes its values in a *referential set*<sup>4</sup>  $I_P$  of intervals. This set is obtained using the constant numerical values of  $P$ . Typically, if  $C$  denotes the set of these constant values, then  $I_P$  is the set of all possible disjoint intervals such that their lower and upper bounds are either  $-\infty$  or  $+\infty$  or elements of  $C$ . For instance, a possible value of  $I_P$  where  $C = \{2, 5, 11\}$  is as follows:

$$I_P = \{ ]-\infty, 2[, [2, 2], ]2, 5[, [5, 5], ]5, 11[, [11, 11], ]11, +\infty[ \}.$$

Since  $P$  may be composed of sub-processes as  $P = P_1 \mid \dots \mid P_k$ , the set  $I_P$ , which is equal to  $I_{P_1 \mid \dots \mid P_k}$ , is obtained from  $C = C_1 \cup \dots \cup C_k$  where each  $C_i$  represents the set of constant values associated with  $P_i$ . In the sequel, a program approximation is determined w.r.t. a given referential set of intervals.

<sup>4</sup> More generally, for a program  $P$ , there could be several referential sets of intervals. Every numerical type (e.g. `integer`, `real`) of constants in  $P$  is associated with a referential set where interval bounds are from the corresponding domain. Here, to simplify, we only consider one referential set of intervals for each program.

#### 4.2.1 Approximation of the primitive constructs

For each primitive construct  $P$  of the language, we define a corresponding abstraction  $\alpha_{I_P}(P)$  on interval domains. Let  $f_a$  be an approximation of a pointwise function  $f$  on intervals:

*Functions/relations:*  $\alpha_{I_P}(y := f(x_1, \dots, x_n)) \rightsquigarrow i_y := f_a(i_{x_1}, \dots, i_{x_n});$

*Delay:*  $\alpha_{I_P}(y := x \$ 1 \text{ init } y_0) \rightsquigarrow i_y := i_x \$ 1 \text{ init } [y_0, y_0];$

*Under-sampling:*  $\alpha_{I_P}(y := x \text{ when } b) \rightsquigarrow i_y := i_x \text{ when } i_b;$

*Deterministic merging:*  $\alpha_{I_P}(y := u \text{ default } v) \rightsquigarrow i_y := i_u \text{ default } i_v;$

*Synchronous composition:*  $\alpha_{I_{P|Q}}(P|Q) \rightsquigarrow \alpha_{I_{P|Q}}(P) | \alpha_{I_{P|Q}}(Q);$

*Hiding:*  $\alpha_{I_P}(P \text{ where } x) \rightsquigarrow \alpha_{I_P}(P) \text{ where } i_x.$

**Example 4.2** Let  $I_P = \{ ]-\infty, 2[, [2, 2], ]2, 5[, [5, 5], ]5, 11[, [11, 11], ]11, +\infty[ \}$  be the set of intervals associated with a program  $P$ . In the following, a stream of  $P$  (on the left) is associated with an abstract stream on  $I_P$  (on the right):

$$\begin{array}{cccccccccccccccc} \mathbf{x} : & 7 & \perp & 4 & 2 & 0 & \dots & & ]5, 11[ & \perp & ]2, 5[ & [2, 2] & ]-\infty, 2[ & \dots \\ \mathbf{b} : & t & \perp & f & f & t & \dots & \Rightarrow & t & \perp & f & f & t & \dots \\ \mathbf{z} : & 5 & \perp & \perp & 11 & 17 & \dots & & [5, 5] & \perp & \perp & [11, 11] & ]11, +\infty[ & \dots \end{array}$$

We denote by  $\mathcal{T}_I$  the set of abstract streams where numerical signals take their values in the set of intervals.

#### 4.2.2 Proof of correctness

Let us define the basic notions on which we have to reason for proving the correctness of the abstraction: the concrete and abstract domains  $\mathcal{C}$  and  $\mathcal{A}$ , the abstraction and concretization functions  $\alpha$  and  $\gamma$ .

- $\mathcal{C}$  is defined as  $\mathcal{P}(\mathcal{T})$ . It admits an *inf* element:  $\emptyset$ . It is associated with the order  $\subseteq$  (inclusion of stream sets).
- $\mathcal{A}$  is defined as  $\mathcal{P}(\mathcal{T}_I)$ . It admits *inf* and *sup* elements, respectively represented by  $\emptyset$  and  $\mathcal{T}_I$ . It is associated with the order  $\subseteq$  (inclusion of stream sets). The union and intersection of stream sets are respectively represented by  $\cup$  and  $\cap$ .
- Abstraction function:

$$\alpha : \mathcal{C} \longrightarrow \mathcal{A}$$

$$c \mapsto a \text{ where } \forall T \in c, \forall T' \in a, \forall t \forall x, T'(t)(x) = i_k \Leftrightarrow T(t)(x) \in i_k$$

- Concretization function:

$$\gamma : \mathcal{A} \longrightarrow \mathcal{C}$$

$$a \mapsto c = \{ T \mid \exists T' \in a \text{ such that } \forall t \forall x, T(t)(x) \in T'(t)(x) \}$$

The following correctness criteria are then verified:

- (i)  $Id \subseteq \gamma \circ \alpha$  or equivalently  $\forall c \in \mathcal{C} \ c \subseteq \gamma(\alpha(c))$ .

$\Rightarrow \forall c \in \mathcal{C}, \exists a \in \alpha(c) \Rightarrow \forall T \in c, \exists T' \in a$  such that  $\exists i_k$  and  $\forall t \forall x T'(t)(x) = i_k \Leftrightarrow T(t)(x) \in i_k$ ; moreover  $c' \in \gamma(a) \Rightarrow c' = \{T \mid \exists T' \in a, \forall t \forall x T(t)(x) \in T'(t)(x)\}$ . Finally, one trivially verifies that  $c \subseteq c'$ , hence  $c \subseteq \gamma(\alpha(c))$ .

(ii)  $Id = \alpha \circ \gamma$  or equivalently  $\forall a \in \mathcal{A} a = \alpha(\gamma(a))$ .

$\Rightarrow \forall a \in \mathcal{A} \exists c = \gamma(a) \Rightarrow c = \{T \mid \exists T' \in a, \forall t \forall x T(t)(x) \in T'(t)(x) \equiv i_k\}$ ; it follows that  $a' \in \alpha(c) \Rightarrow \forall T \in c, \exists T' \in a'$  such that  $\exists i_l$  and  $\forall t \forall x T'(t)(x) = i_l \Leftrightarrow T(t)(x) \in i_l$ . Since considered intervals are disjoint, we have  $i_l = i_k$ . Finally, we obtain that  $a = a'$ , hence  $a = \alpha(\gamma(a))$ .

### 4.3 Translation into IDDs

Let  $P$  be the interval abstraction of a SIGNAL process, we denote by  $\mathcal{T}(P)$  its corresponding translation using IDDs. For each primitive construct of the language, we define its corresponding predicate in terms of IDDs.

*Functions/relations*<sup>5</sup>:  $(y \in I_{f_a}) \wedge (b_{x_1} \Leftrightarrow \dots \Leftrightarrow b_{x_n} \Leftrightarrow b_y)$ ;

*Delay*<sup>6</sup>:  $(y \in i_x \uplus [y0, y0]) \wedge (b_x \Leftrightarrow b_y)$ ;

*Under-sampling*:  $(y \in i_x) \wedge ((b_x \wedge [b]) \Leftrightarrow b_y)$ ;

*Deterministic merging*:  $((y \in i_u \wedge b_u) \vee (y \in i_v \wedge b_v \wedge \neg b_u)) \wedge (b_u \vee b_v \Leftrightarrow b_y)$ ;

*Synchronous composition*:  $\mathcal{T}(\alpha_{I_{p|q}}(P)) \wedge \mathcal{T}(\alpha_{I_{p|q}}(Q))$ ;

*Hiding*:  $\exists x \mathcal{T}(\alpha_{I_p}(P))$ .

The above translation can be applied to a SIGNAL program so that we are able to deal with numerical expressions that define clocks. Typically, condition-clocks are no longer considered as “black boxes” as it was the case in Section 3.2.2. For instance, let us consider again the SIGNAL program of that section (example 3.3). It could be represented as an IDD characterized by the conjunction of the following predicates:

$$(s1) \Rightarrow (x_1 \in I_f) \wedge (b_{x_1} \Leftrightarrow (x \in ]2N, +\infty[))$$

$$(s2) \Rightarrow (x_2 \in I_g) \wedge (b_{x_2} \Leftrightarrow (x \in ]-\infty, N[))$$

The resulting IDD can be now considered to check that the statements **s1** and **s2** cannot satisfy situations where boolean variables  $b_{x_1}$  and  $b_{x_2}$  are both *true*. This information is then used by the compiler to infer the mutual exclusion of the clocks of signals **x1** and **x2**. The SIGNAL clock calculus process cannot determine such a property. As exposed in Section 3.2.2, it mainly focus on a boolean abstraction of the control part of programs, which consists of clocks and boolean conditions. The result of the performed analysis is a clock hierarchy that makes explicit clock inclusions [2]. Clock manipulation algorithms rely on an encoding of clocks into boolean variables. This does not allow to fully reason on condition-clocks defined by numerical expressions. The use of IDDs enables to represent precisely condition-clocks and perform

<sup>5</sup>  $I_{f_a}$  is the interval computed by the interval approximation function  $f_a$  of the pointwise function  $f$  [1].

<sup>6</sup>  $i_x \uplus [y0, y0]$  denotes the smallest interval that contains  $y0$  and all elements of  $i_x$ .

boolean operations that take into account values of condition-clocks. The clock calculus process is therefore able to deal with numerical expressions as well as logical expressions. More program behaviours become analyzable while this is not the case when only considering boolean abstractions during the compilation. In particular, it allows to refine the synchronizability of clocks defined from boolean relations on numerical signals. An additional expected major impact concerns code generation, which should get better optimized.

#### 4.4 *Discussions and related work*

We first observe that IDDs may be used at different description levels of a SIGNAL program. In the above example, the translation is applied to the initial specification of a program. However, it is more interesting to consider descriptions after specific transformations performed by the compiler. Typically, programs that contain extended constructs of the language will necessarily require to be rewritten in the kernel language for which the transformation into IDDs is quite straightforward.

In [22], an extension has been proposed in order to improve the SIGNAL clock calculus. It takes into account the possibility of having affine relations between different clocks in a program. These clocks are therefore said to be synchronizable. A data structure has been implemented, which enables to characterize these relations in addition to the existing internal clock representations within the compiler. In the same spirit, the implementation of our IDD-based translation can extend the clock calculus. Among other related approaches that address the verification of functional properties, we can mention [13] and [14], which use polyhedra techniques to approximate the numerical part. Our approach rather adopts interval techniques similarly to [3], [23] and [10]. While polyhedra generally offer more precise results than intervals when considered as abstract domain, it appears that polyhedra manipulation packages are more complex and may rapidly lead to performance problems (typically, when the number of numerical variables increases).

## 5 Conclusion and future work

In this paper, we propose a simple and promising solution in order to improve the functional analysis of SIGNAL programs. In particular, the solution allows to deal with numerical properties, which are not fully taken into account in the functional analysis performed by the compiler. We consider interval techniques to represent numerical variable domains. These intervals are used in predicates, which are associated with particular data structures called *interval decision diagrams* (IDDs). Similarly to BDDs, IDDs offer a powerful representation allowing one to perform complex boolean operations.

The present work needs to be carried out in POLYCHRONY, the design environment of SIGNAL. All required basic packages already exist. So, the major

efforts rather concern the integration of the packages in the compiler.

## References

- [1] G. Alefeld and J. Hertzberger. *Introduction to Interval Computation*. Academic Press, NY, 1983.
- [2] T.P. Amagbégnon. Forme canonique arborescente des horloges de SIGNAL. In *Thèse de l'Université de Rennes I, IFSIC, France*, December 1995.
- [3] F. Benhamou, L. Granvilliers, and F. Goualard. Interval constraints: Results and perspectives. In *Joint ERCIM/Compulog Net Workshop on New Trends in Constraints*, pages 1–16, London, UK, 2000. Springer-Verlag.
- [4] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. In *Special issue on Embedded Systems, IEEE*, 2003.
- [5] F. Besson, T. Jensen, and J.-P. Talpin. Timed polyhedra analysis for synchronous languages. In *10th International Conference on Concurrency Theory (CONCUR'99), Eindhoven, The Netherlands*, August 1999.
- [6] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE transactions on computers*, C-35(8):677–691, August 1986.
- [7] M. Christiansen and E. Fleury. Interval decision diagram library (libidd). <http://www.cs.aau.dk/~mixxel/libidd/index.html>.
- [8] M. Christiansen and E. Fleury. An MTIDD based firewall using decision diagrams for packet filtering. *Telecommunication Systems, Kluwer Academic Publishers*, 27(2-4):297–319, 2004.
- [9] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic. *ACM Trans. on Programming Languages and Systems*, 2(8):244–263, 1986.
- [10] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.
- [11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *proceedings of POPL'77*, 1977.
- [12] D.A. Duffy. *Principles of automated theorem proving*. John Wiley & Sons, Inc., New York, NY, 1991.
- [13] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.



- [14] B. Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, July 2003.
- [15] M. Kerbœuf, D. Nowak, and J.-P. Talpin. Theorem proving in higher order logics. In *13th International TPHOLs Conference (TPHOLs'2000)*, Portland, Oregon, August 2000.
- [16] M. Kerbœuf, D. Nowak, and J.-P. Talpin. Formal methods and software engineering. In *5th International Conference on Formal Engineering Methods (ICFEM'2003)*, Singapore, November 2003.
- [17] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. In *Journal for Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design. (c)* World Scientific, April 2002.
- [18] J. Lind-Nielsen. Buddy, a Binary Decision Diagram Package. <http://sourceforge.net/projects/buddy>.
- [19] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the SIGNAL environment. In *Discrete Event Dynamic System: Theory and Applications*, 10(4), pages 325–346, Oct. 2000.
- [20] M. Nebut. Réactions synchrones : spécification et analyse. In *Thèse de l'Université de Rennes I, IFSIC, France*, November 2002.
- [21] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction (CADE'92)*, June 1992.
- [22] I. Smarandache. Transformations affines d'horloges: application au codesign de systèmes temps-réel en utilisant les langages SIGNAL et ALPHA. In *Thèse de l'Université de Rennes I, IFSIC, France*, October 1998.
- [23] K. Strehl and L. Thiele. Symbolic model checking of process networks using interval diagram techniques. In *ICCAD*, pages 686–692, 1998.
- [24] The Coq Development Team. The coq proof assistant : Reference manual : Version 7.2. Technical Report 0255, INRIA - Rocquencourt, February 2002.