

Polychronous Design of Embedded Real-Time Applications

Abdoulaye GAMATIÉ

INRIA Futurs

Thierry GAUTIER, Paul LE GUERNIC and Jean-Pierre TALPIN

IRISA/INRIA

The current work has been partially supported by the European project IST SAFEAIR (Advanced Design Tools for Aircraft Systems and Airborne Software) [Goshen-Meskin et al. 2001].

Author's address: INRIA Futurs, Synergie Park, 6bis avenue Pierre et Marie Curie 59260 Lezennes, France.

abdoulaye.gamatie@lifl.fr, {thierry.gautier, paul.le-guernic, jean-pierre.talpin}@irisa.fr

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2005 ACM 0730-0301/2005/0100-0001 \$5.00

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005, Pages 1-0??.

Embedded real-time systems consist of hardware and software along with functional and timing constraints for the interaction with the environment. They are ubiquitous in today’s technological landscape, however, typical domains of embedded real-time systems include telecommunication, nuclear power plants, avionics, and medical technology. These systems are often critical because of the high human and economic stakes. Therefore, the development of such systems requires highly reliable approaches. The synchronous approach is one possible answer for this demand. Its mathematical basis provides formal concepts that favor the trusted design of embedded real-time systems. The multi-clock or *polychronous* model stands out from other synchronous specification models by its capability to allow the design of systems where each component holds its own activation clock as well as single-clocked systems in a uniform way. A great advantage is its convenience for component-based design approaches that allow modular development of increasingly complex modern systems. The expressiveness of its underlying semantics allows to deal with several issues on real-time design.

This article exposes insights gained during recent years from the design of real-time applications within the polychronous framework. In particular, it shows promising results about the design of applications from the avionics domain.

Categories and Subject Descriptors: D.2.1 [**Requirements/Specifications**]: Methodologies; D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering; D.2.4 [**Software/Program Verification**]: Formal methods, Validation; D.4.7 [**Organization and Design**]: Real-time systems and embedded systems

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: Design methodology, synchronous approach, embedded real-time systems, Avionics, formal methods, SIGNAL

1. INTRODUCTION

Embedded real-time systems consist of hardware and software along with functional and timing constraints for the interaction with the environment. They are ubiquitous in today’s technological landscape, however, typical domains of embedded real-time systems include telecommunication, nuclear power plants, avionics, and medical technology. These systems are often critical because of the high human and economic stakes. Therefore, the development of such systems requires highly reliable methodologies. Over the past decade, high-level design of such systems has gained prominence in the face of rising technological complexity, increasing perfor-

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

mance requirements, and tightening time to market constraints. Broad discussions of challenges in the design of these systems can be found in literature [Lee 2000] [Wirth 2001] [Sifakis 2001] [Pnueli 2002]. It is now widely accepted that suitable design frameworks must provide a means to describe systems without ambiguity, to check desired properties of these systems, and to automatically generate code with respect to requirements.

The *synchronous approach* has been proposed in order to answer this demand [Benveniste et al. 2003]. Its basic assumption is that computation and communication are instantaneous. This assumption is often represented by the ideal vision of *zero time* execution (referred to as “synchrony hypothesis”). More precisely, a system is viewed through the chronology and simultaneity of observed events during its execution. This is a main difference from classical approaches where the system execution is rather considered under its chronometric aspect (i.e., duration has a significant role). The mathematical foundations of the synchronous approach provide formal concepts that favor the trusted design of embedded real-time systems.

The multi-clocked or *polychronous* model [Le Guernic et al. 2003] stands out from other synchronous specification models by its capability to allow the design of systems in which each component holds its own activation clock as well as single-clocked systems in a uniform way. A great advantage is its convenience for component-based design approaches that allow modular development of increasingly complex modern systems. As a matter of fact, the design of each component can be addressed separately. The expressiveness of its underlying semantics allows to deal with several issues on real-time design.

A major part of embedded real-time design environments is based on multiple tools for various purposes: specification, verification, evaluation, etc. While such environments are practical, they make it difficult to guarantee the correctness of the designed systems. As a matter of fact, the involved tools generally have different semantic foundations and the translation process of descriptions from one tool

to another becomes error-prone. This leads to a global coherence problem, which affects the description, verification, and validation of systems and represents a big obstacle to the development of safety critical systems such as embedded real-time systems in these environments. One solution to this problem consists in considering the same semantic model for the whole design activity.

In this article, we consider the SIGNAL language, which is based on the polychronous semantic model [Le Guernic et al. 2003] and its associated tool-set POLYCHRONY to design embedded real-time applications. The inherent flexibility of the abstract notions defined in the polychronous framework favors the design of correct-by-construction systems by means of well-defined transformations of system specifications that preserve the intended semantics and stated properties of the system under design. As application domain, we particularly focus on avionics to address modeling and temporal evaluation issues.

Contribution. The first contribution of the work presented in this article is a design approach for embedded real-time applications using the polychronous model. These systems often include asynchronous mechanisms (e.g. to achieve communications). The modeling of such mechanisms using the synchronous paradigm is not trivial. Our design approach is proposed within a homogeneous framework that uses the SIGNAL language as description formalism. As a result, the formal techniques and tools available within the POLYCHRONY platform can be used for system analysis. Targeting avionics, we consider the recent *integrated modular avionics* architectures and their associated standard ARINC [Airlines Electronic Engineering Committee 1997a] [Airlines Electronic Engineering Committee 1997b] to develop a library of so-called APEX-ARINC services, providing SIGNAL models of RTOS functionalities. Then, we show how these models can be used to describe the distributed applications. Finally, we expose a technique for temporal evaluation of these applications using SIGNAL. This allows us to validate our design choices.

Outline. The rest of the article is organized as follows: Section 2 first presents ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

the SIGNAL language and its associated analysis techniques. Then, Section 3 introduces avionic system architectures. Section 4 exposes the modeling of a library of components based on the avionic APEX-ARINC standard. This library is used in Section 5 for the design of distributed applications within POLYCHRONY. Section 6 shows how timing issues can be addressed using a technique based on interpretations of SIGNAL programs. In Section 7, we mention some relevant works related to the study exposed in this article. Finally, concluding remarks are given in Section 8.

2. THE SIGNAL LANGUAGE

The basis of the SIGNAL design is close to the *stream* concept such as defined in the semantics of data-flow languages [Kahn 1974]. The main difference lies in the way streams are built from *traces*. A trace is a stream in which the *bottom* (\perp) value (stating for “no event”) may occur between two defined values. In the following, we present the SIGNAL language and its associated concepts.

2.1 Synchronized data-flow

Consider as an example the following program expressed in some conventional data-flow language:

$$\mathbf{if } a > 0 \mathbf{ then } x = a; y = x + a$$

What is the meaning of this program? In an interpretation where the communication links are considered as FIFO queues [Arvind and Gostelow 1978], if a is a sequence with non-positive values, the queue associated with a will grow forever, or (if a is a finite sequence) the queue associated with x will eventually be empty although a is non-empty. It is not clear that the meaning of this program is the meaning that the author had in mind! Now, suppose that each FIFO queue consists of a single cell [Dennis et al. 1974]. Then as soon as a negative value appears on the input, the execution can no longer go on: there is a deadlock. This is usually represented by the special undefined value \perp .

It would be somewhat significant if such deadlocks could be statically prevented.

For that, it is necessary to be able to statically verify timing properties. Then the \perp should be handled when reasoning about time, but it has to be considered with a non standard meaning. In the framework of synchronized data-flow, the \perp corresponds to the absence of value at a given logical instant for a given variable (or *signal*). In particular, it must be possible to insert several \perp symbols between two defined values of a signal. Such an insertion corresponds to some resynchronization of the signal. However, the main purpose of synchronized data-flow is that the whole synchronization should be completely handled at compile time, in such a way that the execution phase has nothing to do with \perp . This will be assumed by a static representation of the timing relations expressed by each operator. Syntactically, the timing will be implicit in the language. SIGNAL describes processes that communicate through (possibly infinite) sequences of typed values with implicit timing: the *signals*. For example, x denotes the infinite sequence $\{x_t\}_{t \geq 0}$ where t denotes a logical time index. Signals defined with the same time index are said to have the same *clock*, so that clocks are equivalence classes of simultaneous signals. A SIGNAL program consists of a set of processes recursively composed from elementary processes. An elementary process is an expression defining one signal.

Consider a given operator that has, for example, two input signals and one output signal. We shall speak of *synchronous* signals if they are *logically* related in the following sense: for any t , the t^{th} token on the first input is evaluated with the t^{th} token on the second input, to produce the t^{th} token on the output. This is precisely the notion of *simultaneity*. However, for two tokens on a given signal, we can say that one is before the other (*chronology*). Then, for the synchronous approach, an *event* is a set of instantaneous calculations, or equivalently, of instantaneous communications. A *process* is a system of equations over signals. A SIGNAL program is a process.

2.2 SIGNAL constructs

SIGNAL relies on a handful of primitive constructs, which can be combined using a composition operator. These core constructs are of sufficient expressive power to derive other more elaborated constructs for comfort and structuring. Here, we

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

give a sketch of the primitive constructs (bold-faced) and a few derived constructs (italics) often used. For each of them, the corresponding syntax and definition are mentioned.

Functions/Relations. Let f be a symbol denoting a n -ary function $[[f]]$ on instantaneous values (e.g., arithmetic or boolean operation). Then, the SIGNAL expression

$$y := f(x_1, \dots, x_n)$$

defines an elementary process such that:

$$y_t \neq \perp \Leftrightarrow x_{1t} \neq \perp \Leftrightarrow \dots \Leftrightarrow x_{nt} \neq \perp, \forall t : y_t = f(x_{1t}, \dots, x_{nt}),$$

where x_{i_k} denotes the k^{th} element of the sequence denoted by $\{x_{i_t}\}_{t \geq 0}$.

Delay. This operator defines the signal whose t^{th} element is just the $(t - 1)^{th}$ element of its input, at any instant but the first one, where it takes an initialization value. Then, the SIGNAL expression

$$y := x \$ 1 \text{ init } c$$

defines an elementary process such that:

$$x_t \neq \perp \Leftrightarrow y_t \neq \perp, \forall t > 0 : y_t = x_{t-1}, y_0 = c.$$

At the first instant, the signal y takes the initialization value c . Then, at any instant, y takes the previous value of x .

Under-sampling. This operator has one data input and one boolean “control” input, but it has a different meaning when one of the inputs holds \perp . In this case, the output is also \perp ; at any logical instant where both input signals are defined, the output will be different from \perp if and only if the control input holds the value *true*. Then, the SIGNAL expression

$$y := x \text{ when } b$$

defines an elementary process such that:

$$y_t = x_t \text{ if } b_t = \text{true}, \text{ else } y_t = \perp.$$

The derived statement $y := \text{when } b$ is equivalent to $y := b \text{ when } b$.

Deterministic merging. The unique output provided by this operator is defined (i.e., it has a value different from \perp) at any logical instant where at least one of its two inputs is defined (it is not defined otherwise); a priority makes it deterministic. Then, the SIGNAL expression

$$z := x \text{ default } y$$

defines an elementary process such that:

$$z_t = x_t \text{ if } x_t \neq \perp, \text{ else } z_t = y_t.$$

Parallel composition. Resynchronizations (that is to say, possible insertions of \perp) have to take place when composing processes with common signals. However, this is only a formal manipulation. If P and Q denote two processes, the *composition* of P and Q defines a new process, denoted by

$$(| P \ | \ Q \ |)$$

where common names refer to common signals. Then, P and Q communicate through their common signals.

Restriction. This operator allows one to consider as local signals a subset of the signals defined in a given process. If x is a signal defined in a process P,

$$P \text{ where } x$$

defines a new process where signals for external communication with other processes are those of P, except x . Note that when different processes $P_{i,i \in 1..k}$ contain local signals of the same name, these signals are implicitly renamed in the composition of the P_i 's so as to distinguish the local signals in each P_i .

Derived operators are defined from the kernel of primitive operators, for example:

Clock extraction: $h := \hat{x}$ specifies the clock h of x , and can be defined as:

$$h := (x = x).$$

Synchronization: $x1 \hat{=} x2$ specifies that $x1$ and $x2$ have the same clock, and is defined as: $(| h := (\hat{x}1 = \hat{x}2) |)$ where h .

Clock union: $h := x1 \hat{+} x2$ specifies the clock union of $x1$ and $x2$, which is also defined as: $h := \hat{x}1 \text{ default } \hat{x}2$.

Memory: $y := x \text{ cell } b \text{ init } y0$ allows to memorize in y the latest value carried by x when x is present or when b is *true*. It is defined as:

$(| y := x \text{ default } (y\$1 \text{ init } y0) | y \hat{=} x \hat{+} (\text{when } b) |)$.

2.3 A simple example

The purpose of the following process is to define a signal v which counts in the decreasing order the number of occurrences of the events at which a boolean signal *reset* holds the value *false*; v is reinitialized (with a value $v0$) each time *reset* is *true*.

```
(| zv := v $ 1 init 0
  | vreset := v0 when reset
  | zvdec := zv when (not reset)
  | vdec := zvdec - 1
  | v := vreset default vdec
  | reach0 := true when (zv = 1)
  |) where integer zv, vreset, zvdec, vdec;
```

Comments: v is defined with $v0$ each time *reset* is present and has the value *true* (operator *when*); otherwise (operator *default*), it takes the value of $zvdec-1$, $zvdec$ being defined as the previous value of v (delay), zv , when this value is present and moreover, when *reset* is present and has the value *false* (operator *when*). The boolean signal *reach0* is defined (with the value *true*) when the previous value of v is equal to 1. Notice that v is decremented when *reset* has a value *false*.

Program abstraction allows one to declare a given process, together with its ways of communication, stated explicitly. As an example, the above process may be declared as

```
process RCOUNT =
  { integer v0; }
```

```

      ( ? boolean reset;
        ! boolean reach0;
        integer v; )
(| zv := v $ 1 init 0
 | vreset := v0 when reset
 | zvdec := zv when (not reset)
 | vdec := zvdec - 1
 | v := vreset default vdec
 | reach0 := true when (zv = 1)
 |)
where
  integer zv, vreset, zvdec, vdec;
end;

```

It may be referred to as, for example, RCOUNT(10) ($v0$ is a formal parameter of the process; “?” stands as a tag for the input signals and “!” for the output ones). Here, there are one input signal, `reset`, and two output signals, `reach0` and `v`.

2.4 Compiling SIGNAL programs

What are the relevant questions when compiling SIGNAL programs?

- Is the program deadlock free?
- Does it have an effective execution?
- If so, what scheduling may be statically calculated (for a multiprocessor implementation)?

To be able to answer these questions, two basic tools are used before execution on a given architecture. The first one is the modeling of the synchronization relations in the set of polynomials with coefficients in the finite field $\mathbb{Z}/3\mathbb{Z}$ of integers modulo 3 (see Section 2.4.1). In the following, the set $\mathbb{Z}/3\mathbb{Z}$ is also denoted by \mathcal{F}_3 . The second one is the directed graph of data dependencies.

2.4.1 The synchronization space. First, let us consider SIGNAL processes restricted to the single domain of boolean values. The expression

$$x3 := x1 \text{ when } x2$$

expresses the following assertions:

- if x_1 is defined, and x_2 is defined and *true*, then x_3 is defined and $x_1 = x_3$,
- if x_1 is not defined, or x_2 is not defined, or x_2 is defined and *false*, then x_3 is not defined.

It appears that useful informations are (if x is a signal):

- x is defined and *false*,
- x is defined and *true*,
- x is not defined.

They can be respectively encoded in the finite field $\mathbb{Z}/3\mathbb{Z}$ of integers modulo 3 as the following values: $-1, 1$ and 0 . Then, if x is the encoding value associated with the signal x , the presence of the signal x may be clearly represented by x^2 . This representation of an indeterminate value of x (*true* or *false*) leads to an immediate generalization to non boolean values: their presence is encoded as 1 and their absence as 0. In this way, x^2 may be considered as the proper clock of the signal x .

This principle is used to represent synchronization relations expressed through SIGNAL programs. In the following, each signal and its encoding value are denoted by the same variable. The coding of the elementary operators is deduced from their definition. This coding is introduced below:

- The equations

$$y^2 = x_1^2 = \dots = x_n^2$$

denoting the equality of the respective clocks of signals y, x_1, \dots, x_n are associated with $y := f(x_1, \dots, x_n)$ (all the synchronous processes are encoded in this way, however, “dynamical systems” in \mathcal{F}_3 must be used to encode boolean delays - this is not detailed here [Benveniste and Le Guernic 1990]). Boolean relations may be completely encoded in \mathcal{F}_3 . For instance, $x_2 = -x_1$ corresponds to $x_2 := \text{not } x_1$: if $x_1 = \text{true}$, then $x_1 = 1$ and $-(x_1) = -1$, which is associated with *false*.

—The equation

$$x_3 = x_1(-x_2 - x_2^2)$$

is associated with $x_3 := x_1$ **when** x_2 (x_1, x_2, x_3 boolean signals); it may be interpreted as follows: x_3 holds the same value as x_1 ($x_3 = x_1$) when x_2 is *true* (when $-x_2 - x_2^2 = 1$).

—The equation

$$x_3^2 = x_1^2(-x_2 - x_2^2)$$

is associated with $x_3 := x_1$ **when** x_2 when x_1, x_3 are non-boolean signals.

—The equation

$$x_3 = x_1 + (1 - x_1^2)x_2$$

is associated with $x_3 := x_1$ **default** x_2 (x_1, x_2, x_3 boolean signals); it is interpreted as follows: x_3 has a value when x_1 is defined, i.e., when $x_1^2 = 1$ (then x_3 holds the same value as x_1 : $x_3 = x_1^2 x_1 = x_1$), or when x_2 is defined but not x_1 , i.e., when $(1 - x_1^2)x_2^2 = 1$ (then x_3 holds the same value as x_2 : $x_3 = (1 - x_1^2)x_2^2 x_2 = (1 - x_1^2)x_2$).

—The equation

$$x_3^2 = x_1^2 + (1 - x_1^2)x_2^2$$

is associated with $x_3 := x_1$ **default** x_2 when x_1, x_2, x_3 are non-boolean signals.

Then the composition of SIGNAL processes collects the clock expressions of every composing process.

2.4.2 The clock calculus. The algebraic coding of the synchronization relations has a double function. First, it is the way to detect synchronization errors. Consider for example the following program (which is that of Section 2.1):

```
( | c := a>0 | x := a when c | y := x+a | )
```

The meaning of this program is “add a to (a when $a > 0$)”. Its algebraic coding is

$$\begin{aligned}c^2 &= a^2 \\x^2 &= a^2(-c - c^2) \\y^2 &= x^2 = a^2\end{aligned}$$

which results in $c^2 = a^2 = y^2 = x^2 = a^2(-c - c^2)$ and by substitution $c^2 = c^2(-c - c^2)$ and then $c = 1$ or $c = 0$.

This means: c must be either present and *true* or absent. But c is the result of the evaluation of the non boolean signal a . The program induces some constraints on a (or c). Either it can be proved that the environment in which the program will be executed satisfies these constraints and the program is valid, or the environment does not and then the program is rejected.

The other function of this coding is to organize the control of the program. An order relation may be defined on the set of clocks: a clock h^2 is said to be greater than a clock k^2 , which is denoted by $h^2 \geq k^2$, if the set of instants of k is included in the set of instants of h (k is an under-sampling of h). The set of clocks with this relation is a lattice. The purpose of the clock calculus is to synthesize the upper bound of the lattice, which is called the *master clock*, and to define each clock by a calculus expression, i.e., an under-sampling of the master clock according to values of boolean signals. However, for a given SIGNAL process, the master clock may not be the clock of a signal of the process. In this case, several maxima (local master clocks) will be found.

For some processes, the partial order induced by the inclusion of instants, restricted to the under-samplings by a *free* boolean condition (input boolean signal or boolean expression on non-boolean signals), can be a tree, whose root is the most frequent clock. Such processes, also referred to as **endochronous**¹, can be executed in an autonomous way (master mode). Otherwise, when there are several

¹A more formal characterization of **endochrony** can be found in [Le Guernic et al. 2003].

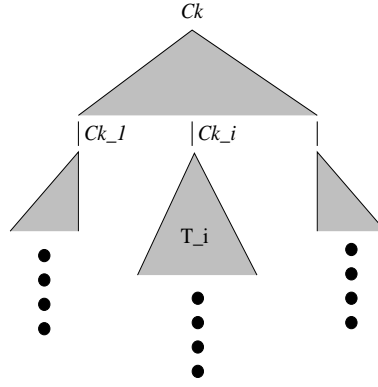


Fig. 1. Clock hierarchy of an endochronous program.

local master clocks in a process, this process needs extra information from its environment in order to be executed in a **deterministic** way. An endochronous process is deterministic [Le Guernic et al. 2003]. FIG. 1 illustrates the clock hierarchy of an endochronous program. It is described by a unique tree where the root node represents the master clock (Ck). We can notice that from this global tree, one can derive several “endochronous” sub-trees (for example T_i).

Clock expressions can be rewritten as boolean expressions of a SIGNAL program. The operator `default` represents the sum of clocks (upper bound) and the operator `when` represents the product (lower bound). Then, any clock expression may be recursively reduced to a sum of monomials, where each monomial is a product of under-samplings (otherwise, the clock is a root).

2.4.3 An example. Consider again the process RCOUNT of Section 2.3. The clock calculus finds the following clocks:

$$\begin{aligned}
 & \text{reset}^2 \\
 \text{vreset}^2 &= -\text{reset} - \text{reset}^2 \\
 \text{v}^2 = \text{zv}^2 = \alpha^2 &= (-\text{reset} - \text{reset}^2) + (\text{reset} - \text{reset}^2)\text{v}^2 \\
 \text{vdec}^2 = \text{zvdec}^2 &= \text{v}^2(\text{reset} - \text{reset}^2) \\
 \text{reach0}^2 &= -\alpha - \text{v}^2
 \end{aligned}$$

where α is the coding of $zv = 1$.

The clock calculus does not synthesize a master clock for this process. In fact, it is not endochronous (and it is non-deterministic): when `reset` is *false*, then `zvdec` is defined if `zv` is defined, i.e., if `v` is defined; but `v` is defined (when `reset` is *false*) if `vdec` is defined, i.e., if `zvdec` is defined, and then, when `reset` is *false*, an occurrence of `v` may occur, but does not necessarily occur.

The hierarchy of clocks is represented by the following SIGNAL process, which defines several trees (whose roots are `ck_1`, `ck_2` and `ck_3`):

```
(| (| ck_1 ^= reset
  | (| ck_1_1 := true when reset
    | ck_1_1 ^= vreset
    | ck_1_2 := true when (not reset)
    |)
  |)
| (| ck_2 := ck_1_2 when ck_3
  | ck_2 ^= vdec ^= zvdec
  |)
| (| ck_3 := ck_1_1 default ck_2
  | ck_3 ^= v ^= zv}
  | (| ck_3_1 := true when(zv=1)
    | ck_3_1 ^= reach0
    |)
  |)
|)
```

The hierarchy is syntactically represented here by the composition embeddings. The variables `ck_i` represent names of the clocks considered as signals (the numbers i are given by the compiler).

Now, we consider the following process, where `RCOUNT` is used in some context

```

process USE_RCOUNT =
  { integer v0; }
  ( ? boolean h;
    ! boolean reach0;
    integer v; )
(| h ^= v
 | reset := (^reach0 when (^h)) default (not (^h))
 | RCOUNT (v0)
 |)
where
  event reset;
end

```

An external clock h defines the instants at which v has a value. The `reset` signal is also synchronous with h and it has the value *true* exactly when `reach0` is present. There is a master clock ($h^2 = v^2 = \text{reset}^2$) and a tree may be built by the compiler. Therefore, the program becomes endochronous.

2.4.4 *The graph of conditional dependencies.* The second tool necessary to implement a SIGNAL program on a given architecture is the graph of data dependencies. Then, according to criteria to be developed, it will be possible to define subgraphs that may be distributed on different processors. However, a classical data-flow graph would not really represent the data dependencies of a SIGNAL program. Since the language handles signals whose clocks may be different, the dependencies are not constant. For that reason, the graph has to express *conditional* dependencies, where the conditions are nothing but the clocks at which dependencies are effective. Moreover, in addition to dependencies between signals, the following relation has to be considered: for any signal x , the values of x cannot be known before its clock; in other words, x depends on x^2 . This relation is assumed implicitly below.

The *Graph of Conditional Dependencies* (GCD) calculated by the SIGNAL compiler for a given program is a labeled directed graph where

- the vertices are the signals, plus clock variables,
- the arcs represent dependence relations,
- the labels are polynomials on \mathcal{F}_3 which represent the clocks at which the relations are valid.

The following describes the dependencies associated with elementary processes. The notation $c^2 : x_1 \rightarrow x_2$ means that x_2 depends on x_1 exactly when $c^2 = 1$. Then, we consider only processes defining non-boolean signals:

$$y := f(x_1, \dots, x_n) \qquad y^2 : x_1 \rightarrow y, \dots, \quad y^2 : x_n \rightarrow y$$

$$y := x \text{ when } b \qquad y^2 : x \rightarrow y, \quad y^2 : b \rightarrow y^2$$

$$z := x \text{ default } y \qquad x^2 : x \rightarrow z, \quad y^2 - x^2 y^2 : y \rightarrow z$$

Notice that the delay does not produce data dependencies (nevertheless, remember that any signal is preceded by its clock).

The graph, together with the clock calculus, is used to detect incorrect dependencies. Such a bad dependency will appear as a cycle in the graph. However, since dependencies are labeled by clocks, some cycles may not occur at any time. An effective cycle is such that the product of the labels of its arcs is not null. This may be compared with the cycle sum test of [Wadge 1979], to detect deadlock on the dependence graph of a data-flow program.

All the above properties checked by the SIGNAL compiler during the clock calculus are mainly static. For more information on issues related to the clock calculus, the reader may refer to [Amagbegnon et al. 1994] and [Amagbegnon et al. 1995], which discuss in depth the capabilities of the compiler to address typical SIGNAL programs. Dynamic properties such as reachability or liveness cannot be addressed by the compiler. For that, the tool SIGALI, which implements a symbolic model checking technique can be used [Marchand et al. 2000]. Basically, a SIGNAL program denotes an automaton in which states are described by the so-called “state variables” that are defined by the *delay* operator. At each logical instant, the current state of

a program is given by the current values of its state variables. The technique adopted in SIGALI consists in manipulating the system of equations resulting from the modeling of SIGNAL programs in \mathcal{F}_3 instead of the sets of its states. This avoids the enumeration of the state space, which can potentially explode. So, each set of states is uniquely characterized by a predicate and operations on sets can be equivalently performed on the associated predicates. Experiments show that the symbolic model-checking technique adopted by SIGALI enables to check properties on automata with several millions of states within a reasonable delay. More details on SIGALI can be found in [Marchand et al. 2000]. Finally, some case studies using SIGALI for verification can be found in [Benveniste et al. 2002] and [Gamatié and Gautier 2003a].

2.5 Temporal analysis of SIGNAL programs

A technique has been defined in order to address timing issues of SIGNAL programs on different implementation platforms [Kountouris and Le Guernic 1996]. Basically, it consists of formal transformations of a program into another SIGNAL program that corresponds to a so-called *temporal interpretation* of the initial one. The new program $\mathcal{O}(P)$ serves as an *observer* of the initial program P in which we only specify the properties we want to check. In particular, we are interested in temporal properties. As shown in Figure 2, the observer receives from the observed program the signals required for analysis and indicates whether or not the considered properties have been satisfied (this can be expressed, e.g., through boolean output signals like in LUSTRE programs [Halbwachs et al. 1993]). The use of observers for verification is very practical because they can be easily described in the same formalism as the observed program. Thus, there is no need to combine different formalisms as in other analysis techniques such as some model-checking techniques, which associate temporal logics with automata [Daws and Yovine 1995]. In Section 6, we present in a detailed way how the temporal interpretation $\mathcal{O}(P)$ of a program P is defined and used for temporal evaluation.

The POLYCHRONY environment [ESPRESSO-IRISA.] associated with the SIGNAL language offers several functionalities including all the facilities mentioned in ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

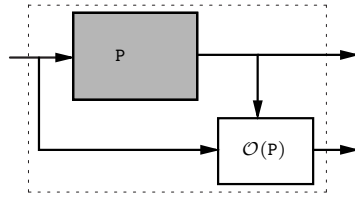


Fig. 2. Composition of a program P together with its observer $\mathcal{O}(P)$.

the above sections.

3. AVIONIC ARCHITECTURES

Traditional architectures in avionic systems are called *federated* [Airlines Electronic Engineering Committee 1997a] [Rushby 1999]. Functions with different criticality levels are hosted on different fault-tolerant computers. FIG. 3 illustrates such an architecture where n functions are considered. A great advantage of such architectures is fault containment. However, this potentially leads to massive usage of computing resources since each function may require its dedicated computer. Consequently, maintenance costs can increase rapidly.



Fig. 3. Federated architectures: each avionics function has its own fault-tolerant computers.

Integrated Modular Avionics (IMA). The recent IMA architectures propose a new way to deal with major obstacles inherent to federated architectures [Airlines Electronic Engineering Committee 1997a]. In IMA, several functions with possibly different criticality, are allowed to share the same computer resources (see FIG. 4).

They are guaranteed a safe allocation of shared resources so that no fault propagation occurs from one component to another component. This is achieved through a *partitioning* of resources with respect to available time and memory capacities.

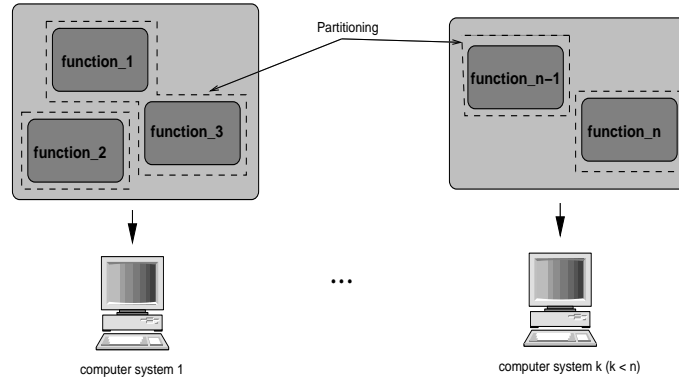


Fig. 4. Integrated Modular Avionics: different functions can share a fault-tolerant computer.

A *partition* is a logical allocation unit resulting from a functional decomposition of the system. IMA platforms consist of *modules* grouped in *cabinets* throughout the aircraft. A module can contain several partitions that possibly belong to functions of different criticality levels. Mechanisms are provided in order to prevent a partition from having “abnormal” access to the memory area of another partition. A processor is allocated to each partition for a fixed time window within a major time frame maintained by the *module-level OS*. A partition cannot be distributed over multiple processors either in the same module or in different modules. Finally, partitions communicate asynchronously via logical *ports* and *channels*. Message exchanges rely on two transfer modes: *sampling* mode and *queuing* mode. In the former, no message queue is allowed. A message remains in the source port until it is transmitted via the channel or it is overwritten by a new occurrence of the message. A received message remains in the destination port until it is overwritten. A refresh period attribute is associated with each sampling port. When reading a port, a *validity* parameter indicates whether the age of the read message is consistent with the required refresh period attribute of the port. In the queuing mode,

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

ports are allowed to store messages from a source partition in queues until they are received by the destination partition. The queuing discipline for messages is First-In First-Out (FIFO).

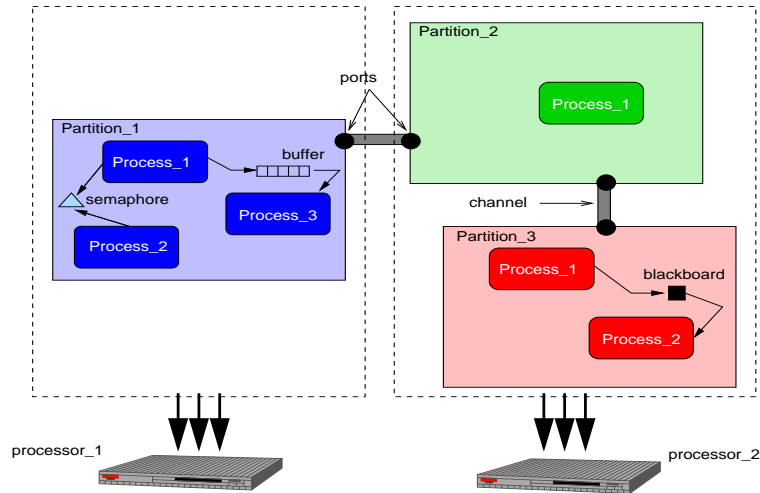


Fig. 5. An example of application partitioning.

Partitions are composed of *processes* that represent the executive units². Processes run concurrently and execute functions associated with the partition in which they are contained. Each process is uniquely characterized by information (like its period, priority, or deadline time) useful to the *partition-level OS* which is responsible for the correct execution of processes within a partition. The scheduling policy for processes is priority preemptive. Communications between processes are achieved by three basic mechanisms. The bounded *buffer* allows to send and receive messages following a FIFO policy. The *event* permits the application to notify processes of the occurrence of a condition for which they may be waiting. The *blackboard* is used to display and read messages: no message queues are allowed, and any message written on a blackboard remains there until the message is either cleared or overwritten by a new instance of the message. Synchronizations are achieved

²An IMA partition/process is akin a UNIX process/task.

using a *semaphore*. FIG. 5 illustrates an example of IMA partitioning.

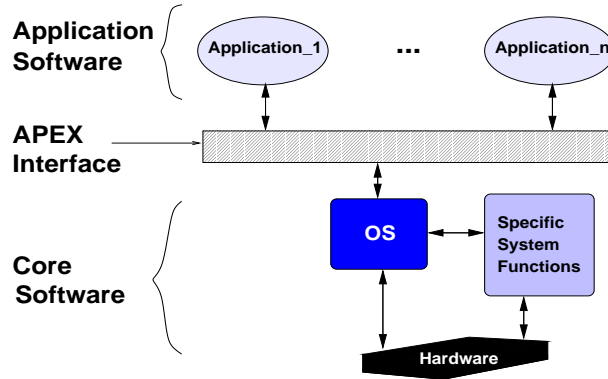


Fig. 6. The APEX interface within the core module Software.

Several standards for software and hardware have been defined for IMA. Here, we particularly concentrate on the APEX-ARINC 653 standard [Airlines Electronic Engineering Committee 1997b], which proposes an OS interface for IMA applications, called *Avionics Application Software Standard Interface* (see FIG. 6). It includes services for communication between partitions on the one hand and between processes on the other hand, synchronization services for processes, partition and process management services, and time and error management services.

4. MODELING OF ARINC CONCEPTS IN SIGNAL

The polychronous design of avionic applications relies on a few basic blocks [Gamatié and Gautier 2003b], which allow us to model partitions:

- (1) APEX-ARINC 653 services (they describe communication and synchronization, partition, process and time management...);
- (2) an RTOS model (it is partially described using complementary services providing functionalities that are not provided via APEX-ARINC 653 services, such as process scheduling);
- (3) executive entities (they mainly consist of generic process models).

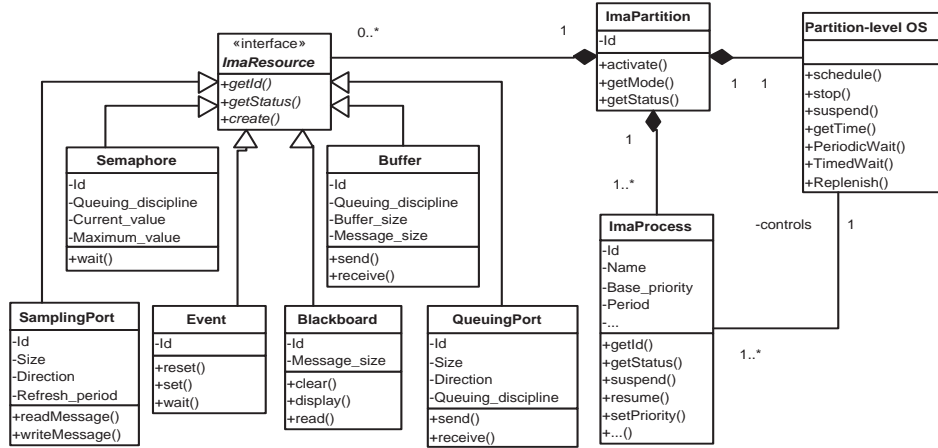


Fig. 7. Basic building blocks for partition modeling.

These building blocks are summarized in the UML diagram illustrated in FIG. 7. In the following, we show for each block, the way its corresponding SIGNAL model is obtained.

4.1 APEX services

The modeling approach adopted here is illustrated by considering a typical APEX service: the *read_blackboard* service. It enables messages to be displayed and read in a blackboard. Input parameters are the blackboard *identifier* and a *time-out* duration (that limits the waiting time on a request, when the blackboard is empty). Output parameters are a *message* (defined by its address and size) and a *return code* (for the diagnostics of the service request). A typical informal specification of the service [Airlines Electronic Engineering Committee 1997b] is given in FIG. 8.

In the following, the modeling of the *read_blackboard* service is presented through different steps. From one step to another, we progressively detail the service description in SIGNAL (here, we only discuss two steps).

A first SIGNAL description of *read_blackboard*. FIG. 9 shows an abstract formal specification corresponding to the service. This specification mainly expresses

```

If inputs are invalid (that means the blackboard identifier is
    unknown or the time-out value is ‘‘out of range’’) Then
    Return INVALID_PARAM (as return code);
Else If a message is currently displayed on the specified
    blackboard Then
    send this message and return NO_ERROR;
Else If the time-out value is zero Then
    Return NOT_AVAILABLE;
Else If preemption is disabled or the current process is the error
    handler Then
    Return INVALID_MODE;
Else
    set the process state to waiting;
    If the time-out value is not infinite Then
        initiate a time counter with duration time-out;
    EndIf;
    ask for process scheduling (the process is blocked and will
        return to ‘‘ready’’ state by a display service request on
        that blackboard from another process or time-out
        expiration);
    If expiration of time-out Then
        Return TIMED_OUT;
    Else
        the output message is the latest available message
        of the blackboard;
        Return NO_ERROR;
    EndIf;
EndIf

```

Fig. 8. An informal specification of the `read_blackboard` service.

interface properties. For example, (s.2) specifies logical instants at which a return code is produced. The variable `C_return_code` is a local boolean signal that carries the value *true* whenever a return code is received on a read request (in other words, `C_return_code` represents the clock of the return code signal). Indeed, on a *read_blackboard* service request, a return code retrieval is not systematic. For instance, when the *blackboard* is empty and the value of the input parameter

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

`timeout` is not infinite (represented in [Airlines Electronic Engineering Committee 1997b] by a special constant *INFINITE_TIME_VALUE*), the requesting process is suspended. In this case, *C_return_code* carries the value *false*. The suspended process must wait: either a message is displayed on the *blackboard*, or the expiration of the active time counter initialized with `timeout` (hence, the return code carries the value *TIMED_OUT*). For the moment, `C_return_code` appears in the *read_blackboard* description as a local signal. It will be defined during refinements of this *read_blackboard read_blackboard read_blackboard read_blackboard read_blackboard* abstract description. At this stage, we assume that there only exist signals such that properties in which it is involved are satisfied. Property (s.1) states that `C_return_code` and all input parameters are synchronous (i.e., whenever there is read request, `C_return_code` indicates whether or not a return code should be produced). Property (s.3) expresses the fact that messages are received on a read request only when the return code value is *NO_ERROR*.

Lines (d.1) and (d.2) give dependency relations between input and output parameters. In *SIGNAL*, the notation $x \dashrightarrow y$ expresses a dependency relation between two signals x and y within a logical instant (y is also said to be preceded by x). For instance, (d.2) states that `message` and `length` are preceded by `timeout` and `board_ID`, at the logical instants where the return code carries the value *NO_ERROR*.

The level of detail provided by a description like the one given in FIG. 9 is expressive enough to check, for instance, the conformance of a component model during its integration in a system described in the same formalism. Here, the description exhibits the interface properties of the *read_blackboard* service. In particular, it gives conditions that describe when a message is received by a process on a read request. However, the description does not specify exactly **how** messages are obtained.

The specifications given in [Airlines Electronic Engineering Committee 1997b] are sometimes imprecise. As a result, this leads to ambiguities, which are not

```

process READ_BLACKBOARD =
  { ProcessID_type process_ID; }
  ( ? Comm_ComponentID_type board_ID;
    SystemTime_type timeout;
    ! MessageArea_type message;
    MessageSize_type length;
    ReturnCode_type return_code; )
  ( | ( | { {board_ID, timeout} -->
          return_code } when C_return_code      (d.1)
    | { {board_ID, timeout} --> {message, length} }
      when (return_code = #NO_ERROR)           (d.2)
    | )
  | ( | board_ID ^= timeout ^= C_return_code   (s.1)
    | return_code ^= when C_return_code        (s.2)
    | message ^= length ^= when (return_code = #NO_ERROR) (s.3)
    | )
  | ) where boolean C_return_code

```

Fig. 9. Abstract description of the *read_blackboard* service .

easily perceptible. Here, two possible implementations are distinguished for the *read_blackboard* service. They mainly depend on the interpretation of message retrieval. Let us consider a process P_1 , which was previously blocked on a read request in a blackboard, and now released on a display request by another process P_2 :

- (1) some implementations assume that the message read by P_1 (the suspended process) is the same as the one just displayed on the blackboard P_2 ;
- (2) there are other implementations that display the message retrieved by P_1 *when the execution of P_1 gets resumed* (as a matter of fact, a higher priority process could be ready to execute when P_1 gets released). So, P_1 will not necessarily read the message displayed by P_2 since the message may have been overwritten when its execution is resumed.

The level of detail of the model described in FIG. 9, although very abstract,
 ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

allows to cover both interpretations of the *read_blackboard* service. In practice, we observe that these interpretations can be useful depending on the context.

- Implementations of type (1) may be interesting when all the messages displayed on the blackboard are relevant to the process P_1 . Every message must be retrieved. However, even if using a blackboard for such message exchanges appears cheaper than using a buffer (in terms of memory space required for message queuing, and of blocked processes management), it would be more judicious to consider a buffer for such communications since it prevents the loss of messages.
- On the other hand, implementations of type (2) find their utility when P_1 does not need to retrieve all displayed messages. For instance, P_1 only needs to read *refreshed data* of the same type. In that case, only latest occurrences of messages are relevant.

The presence of ambiguities as illustrated above justifies a model refinement design approach: abstract descriptions are progressively refined in order to derive particular implementations. Here, the way messages are retrieved during the *read_blackboard* service call is not fixed (there are two possibilities). The SIGNAL specification given in FIG. 9 captures such a situation.

In the next step, the more general model defined previously is refined by precisising more properties in the SIGNAL specification. In particular, we make a choice on how messages are retrieved: the second interpretation.

A more detailed SIGNAL description of *read_blackboard*. A more detailed version of the service is now illustrated in FIG. 10. It is represented by a graphical description of the service defined using the POLYCHRONY graphical user interface. In this figure, we did not report the interface properties specified at the previous step. However, they are still considered. So, the model in FIG. 10 gives more precisions on how the service model is defined. In other words, internal properties can now be specified in addition to interface properties.

We consider a decomposition that allows us to separate concerns. Here, four

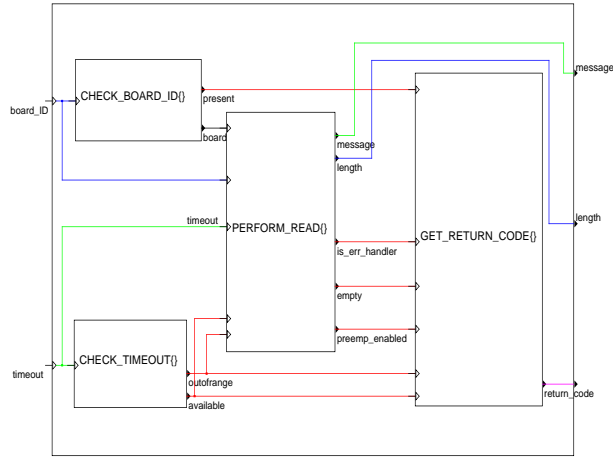
main sub-parts are distinguished based on the service informal specification. they are represented by inner boxes in the graphical description (see FIG. 10). The sub-parts `CHECK_BOARD_ID` and `CHECK_TIMEOUT` verify the validity of input parameters `board_ID` and `timeout`. If these inputs are valid, `PERFORM_READ` tries to read the specified blackboard. Afterward, it has to send the latest message displayed on the blackboard. The area and size of the message are specified by `message` and `length`. `PERFORM_READ` also transmits all the necessary information to `GET_RETURN_CODE`, which defines the final diagnostic message of the service request.

At this stage, we have specified global interface properties and identified the different sub-parts of the service model. Each sub-part is only characterized by its interface. Its internal properties are not yet defined. We can specify some relations between the interface signals of the identified sub-parts. For instance, this is what equations (s1), (s2), (s3), (s4) and (s5) express by synchronizing some signals. In addition to these equations, we can also define the boolean signal `C_return_code`, which was only declared at the previous step. As a matter of fact, we have now enough information to be able to determine its values. This is described by the equation (s6). So, logical instants where `C_return_code` carries the value *true* denote the presence of a return code.

In a similar way, we can specify the internal properties of each sub-part. Finally, a complete SIGNAL specification of the service is obtained as illustrated in [Gamatié and Gautier 2002]. The other APEX services are modeled following the same approach as the *read_blackboard* service. One major advantage of such an approach is that we are able to specify complex programs without omitting any behavioral aspect of a program. Furthermore, the different levels of specification resulting from each step may be used for various purpose as discussed in the first step.

The modeled APEX-ARINC 653 services can be used to describe process management, communication and synchronization between processes, etc. The next section

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.



```

(| board_ID ^= timeout ^= present ^= outofrange ^= available ^=
    C_return_code                                     (s1)
| board ^= empty ^= when present                    (s2)
| message ^= length ^= when (not empty)              (s3)
| is_err_handler ^= when empty when available         (s4)
| preemp_enabled ^= when (not is_err_handler)         (s5)
| C_return_code := (when ((not present) or outofrange)) default
    (when empty when (not available)) default
    (when ((not preemp_enabled) default is_err_handler)) default
    (when (not empty)) default
    false                                           (s6)
| return_code ^= when C_return_code                  (s7)
|)

```

Fig. 10. Refined description of the *read_blackboard* service and clock relations between signals.

presents the modeling of the *partition-level OS*, which is in charge of controlling the execution of processes within a partition.

4.2 Partition-level OS

The role of the partition level OS is to ensure the correct concurrent execution of processes within the partition (each process must have exclusive control on the processor). A sample model of the partition level OS is depicted in FIG. 11.



Fig. 11. Interface of the partition level OS model.

The notions taken into account for the modeling of the *partition level OS* are mainly: *process management* (e.g. create, suspend a process), *scheduling* (including the definition of process descriptors and a scheduler), *time management* (e.g. update time counters), *communications*, and *synchronizations* between processes. The APEX interface provides a major part of required services to achieve the notions mentioned above. However, in order to have a complete description of the *partition level OS* functionalities, we added additional services to our library. These services allow us to describe process scheduling within a partition and they also allow to update time counters. Their description can be found in [Gamatié and Gautier 2003b]. A case study using these services is presented in [Gamatié et al. 2004]. Here, we only present the generic interface of the *partition level OS* (cf. FIG. 11). We explain how it interacts with processes within a partition.

In FIG. 11, the input `Active_partition_ID` represents the identifier of the running partition selected by the module-level OS, and it denotes an execution order when it identifies the current partition (the activation of each partition depends on this signal. It is produced by the module-level OS, which is in charge of the management of partitions in a module). The presence of the input signal `initialize` corresponds to the initialization phase of the partition. It comprises the creation of all mechanisms and processes contained in the partition. Whenever the partition executes, the `PARTITION_LEVEL_OS` selects an active process within the partition. The process is identified by the value carried by the output signal `Active_process_ID`, which is sent to each process. The signal `dt` denotes duration information corresponding to process execution (more precisely, the duration

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

of the current “block” of actions executed by an active process, see Section 4.3 for more details). It is used to update time counter values. The signal `timeout` produced by the partition-level OS carries information about the current status of the time counters used within the partition. For instance, a time counter is used for a wait when a process gets interrupted on a service request with time-out. As the partition-level OS is responsible for the management of time counters, it notifies each interrupted process of the partition with the expiration of its associated time counter. This is reflected by the signal `timeout`.

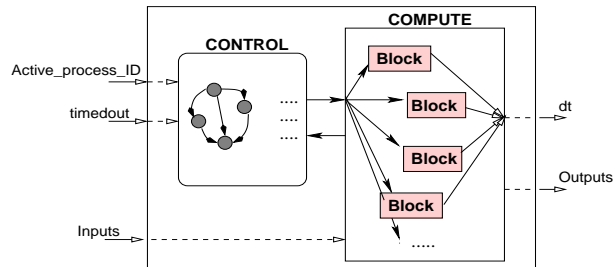


Fig. 12. ARINC process model.

4.3 IMA processes

The definition of an IMA process model basically takes into account its computation and control parts. This is depicted in FIG. 12. Two sub-components are clearly distinguished within the model: *CONTROL* and *COMPUTE*. Any process is seen as a reactive component, which reacts whenever an execution order (denoted by the input `Active_process_ID`) is received. The input `timeout` notifies processes of time-out expiration, while the output `End_Processing` is emitted by the process after completion. In addition, there are other inputs (respectively outputs) needed for (respectively produced by) the process computations. The *CONTROL* and *COMPUTE* sub-components cooperate to achieve the correct execution of the process model.

The *CONTROL* sub-component specifies the control part of the process. Basically, it is a transition system that indicates which statements should be executed

when the process model reacts. It can be encoded easily by an automaton in SIGNAL. Whenever the input `Active_process_ID` (of numeric type) identifies the IMA process, this process “executes”. Depending on the current state of the transition system representing the execution flow of the process, a *block* of actions in the COMPUTE sub-component is selected to be executed *instantaneously* (this is represented by the arrow from CONTROL to COMPUTE in the figure).

The COMPUTE sub-component describes the actions computed by the process. It is composed of *blocks* of actions. They represent elementary pieces of code to be executed without interruption. The statements associated with a block are assumed to *complete within a bounded amount of time*. In the model, a block is executed instantaneously. Therefore, one must take care of what kinds of statements can be put together in a block. Two sorts of statements are distinguished. Those which may cause an interruption of the running process (e.g. a *read_blackboard* request) are called *system calls* (in reference to the fact that they involve the partition level OS). The other statements are those that never interrupt a running process, typically data computation functions. They are referred to as *functions*.

The process model proposed here is very simple. However, for a correct execution, we suggest that at most one system call can be associated with a block, and no other statement should follow this system call within the same block. As a matter of fact, since a block is executed instantaneously, what would happen if the system call interrupts the running process? Typically it is not required to re-execute the whole block when the process gets resumed. Here, all other statements within the block would be executed in spite of the interrupt, and this would not be correct. On the other hand, we note that the suggested restriction can be avoided by using appropriate control mechanisms of SIGNAL. Typically, a boolean state variable could be introduced within each block. Then, the execution of statements would be under-sampled by values of this variable. This solution potentially increases the number of state variables in a program. It is therefore expensive. One can imagine further possibilities but for the sake of simplicity, we rather advocate to avoid the

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

combination of system calls and other statements in the same block.
rtcsa:gamatie.

4.4 Partitions

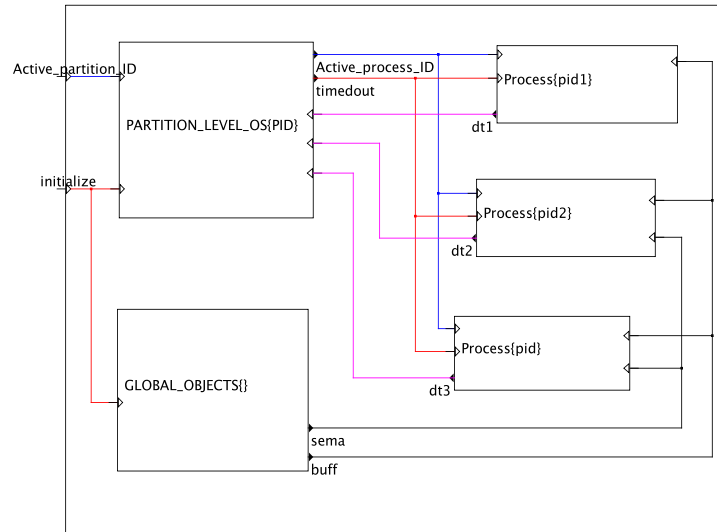


Fig. 13. An example of partition model composed of three processes.

FIG. 13 roughly shows a global view of a partition composed of three processes. In this model, the component `GLOBAL_OBJECTS` appears for structuring. In particular, communication and synchronization mechanisms used by processes (e.g. `buff`, `sema`) are created there.

The UML sequence diagram³ depicted in FIG. 14 illustrates how the partition-level OS interacts with a process during the execution of the partition. After the initialization phase, the partition gets activated (i.e. when receiving *Active_partition_ID*). The partition-level OS selects an active process within the partition. Then, the CONTROL subpart of each process checks whether or not the concerned process can execute. In the diagram, this is denoted by the *optional* action (represented by a box labeled *opt*). When a process is designated by the OS, this

³UML Specification version 2.0: Superstructure – *Object Management Group* (www.omg.org).

action is performed: the process executes a block from its COMPUTE subpart, and the duration corresponding to the executed block is returned to the partition-level OS in order to update time counters. The execution of the model of the partition follows this basic pattern until the module-level OS selects a new partition to execute.

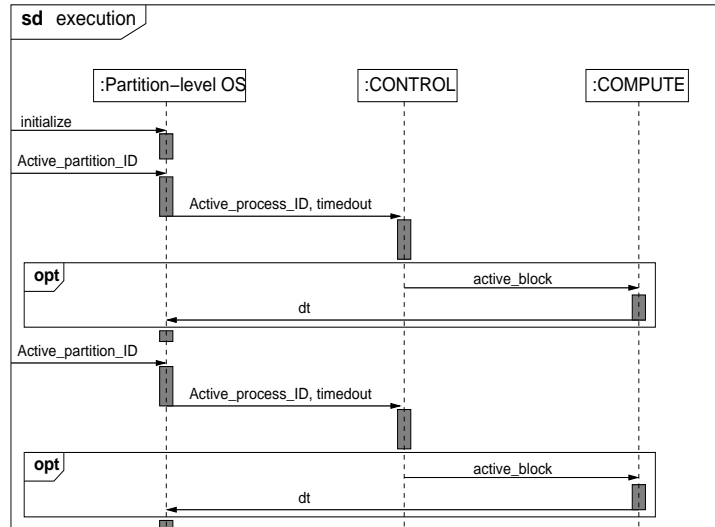


Fig. 14. A sketch of the model execution.

5. DESIGNING APPLICATIONS USING ARINC CONCEPTS MODELS

We now present an approach where the ARINC concepts previously modeled are used to design distributed applications. The approach consists of a set of transformations that, starting from an initial description P (a SIGNAL program), progressively define less abstract SIGNAL programs in the following way: at each step, a new description Q is obtained through the “instantiation” of intermediate variables by adding equations to P . These transformations modify non functional properties of P (e.g. temporal properties by introducing delays during the execution or by relaxing some synchronization relations), but its functional properties are strictly preserved.

5.1 Preliminary notions

The notions presented below have been introduced during the European project SACRES [Gautier and Le Guernic 1999] whose goal was to define ways for generating distributed code from synchronous specifications (particularly SIGNAL programs). Further technical details on this topic can be found in [Benveniste 1998].

In the following, an application is represented by a SIGNAL program

$$P = P_1 \mid P_2 \mid \dots \mid P_n$$

where each sub-program P_i can be itself recursively composed of other sub-programs (i.e., $P_i = P_{i1} \mid P_{i2} \mid \dots \mid P_{im}$). The following hypotheses are assumed:

- (1) considered programs P are *endochronous* (see Section 2.4.2), hence deterministic.
- (2) they do not contain any definition leading to cycles;
- (3) a set of processors $q = \{q_1, q_2, \dots, q_m\}$; and
- (4) a function $locate : \{P_i\} \rightarrow \mathcal{P}(q)$, which associates with each subpart of an application $P = P_1 \mid P_2 \mid \dots \mid P_n$ a non empty set of processors (the allocation can be done either manually or automatically).

First transformation. Let us consider a SIGNAL program $P = P_1 \mid P_2$, as illustrated in FIG. 15. Each sub-program P_i (represented by a circle) is itself composed of four sub-programs P_{i1}, P_{i2}, P_{i3} and P_{i4} . The program P is distributed on two processors q_1 and q_2 as follows:

$$\begin{aligned} \forall i \in \{1, 2\} \forall k \in \{1, 2\}, \text{ locate}(P_{ik}) &= \{q_1\}, \text{ and} \\ \forall i \in \{1, 2\} \forall k \in \{3, 4\}, \text{ locate}(P_{ik}) &= \{q_2\} \end{aligned}$$

Hence, P can be rewritten into $P = Q_1 \mid Q_2$, where $Q_1 = P_{11} \mid P_{12} \mid P_{21} \mid P_{22}$ and $Q_2 = P_{13} \mid P_{14} \mid P_{23} \mid P_{24}$:

$$\begin{aligned}
P &= P_1 \mid P_2 \\
&= (P_{11} \mid P_{12} \mid P_{13} \mid P_{14}) \mid (P_{21} \mid P_{22} \mid P_{23} \mid P_{24}) \\
&= (P_{11} \mid P_{12}) \mid (P_{13} \mid P_{14}) \mid (P_{21} \mid P_{22}) \mid (P_{23} \mid P_{24}) \\
&= (P_{11} \mid P_{12}) \mid (P_{21} \mid P_{22}) \mid (P_{13} \mid P_{14}) \mid (P_{23} \mid P_{24}) \text{ (commutativity of } \mid \text{)} \\
&= (P_{11} \mid P_{12} \mid P_{21} \mid P_{22}) \mid (P_{13} \mid P_{14} \mid P_{23} \mid P_{24}) \\
&= Q_1 \mid Q_2
\end{aligned}$$

Remark 5.1. The above transformation remains valid even if $locate(P_{ik})$ is not a singleton. In that case, P_{ik} is split into new sub-programs which are considered at the same level as P_{jl} 's where $locate(P_{jl})$ is a singleton. For instance, let us consider the program P , it can be rewritten as:

$$P = P_{11..13} \mid P_{12} \mid P_{14} \mid P_{21} \mid P_{22} \mid P_{23} \mid P_{24}$$

where $locate(P_{11..13}) = \{q_1, q_2\}$. Then it follows that

$$\begin{aligned}
P &= P_{11} \mid P_{13} \mid P_{12} \mid P_{14} \mid P_{21} \mid P_{22} \mid P_{23} \mid P_{24} \text{ (} P_{11..13} \text{ is split)} \\
&= P_{11} \mid P_{12} \mid P_{13} \mid P_{14} \mid P_{21} \mid P_{22} \mid P_{23} \mid P_{24} \text{ (commutativity of } \mid \text{)} \\
&= P_1 \mid P_2
\end{aligned}$$

Finally, one can easily derive Q_1 and Q_2 from P .

The sub-programs Q_1 and Q_2 resulting from the partitioning of P are called *s-tasks* [Gautier and Le Guernic 1999]. This transformation yields a new form of the program P that reflects a multi-processor architecture. It also preserves the semantics of the transformed program (since it simply consists of program rewrite).

Remark 5.2. A critical question in the above transformation is how hidden signals are handled. The obvious concern is name capture. For instance, let us consider the example from FIG. 15, if the channel between P_{11} and P_{13} was hidden, and P_{21} has a signal of the same name, there could be a conflict when P_{11} and P_{21} are in the same process Q_1 . In fact, this situation can never happen here since local signals of the same name, from different processes, are implicitly renamed in the ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

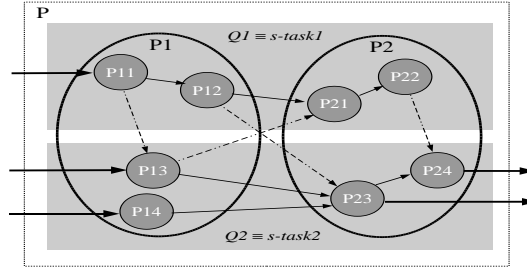


Fig. 15. Decomposition of a SIGNAL process into two s-tasks Q_1 and Q_2 .

composition of such processes (see Section 2.2).

Second transformation. We want to refine the level of granularity resulting from the above transformation. For that, let us consider descriptions at processor level (in other words, s-tasks). We are now interested in how to decompose s-tasks into fine grain entities. An s-task can be seen as a set of *nodes* (e.g. P_{11}, P_{12}, P_{21} and P_{22} in Q_1). In order to have an optimized execution at the s-task level, nodes are gathered in such a way that they can be executed atomically. By atomic execution, we mean nodes execution completes in its entirety without interruption. So, we distinguish two possible ways to define such subsets of nodes, also referred to as *clusters*: either they are composed of a single SIGNAL primitive construct, or they contain more than one primitive construct. The former yields a finer granularity than the latter. However, from the execution point of view, the latter is more efficient since more actions can be achieved at a same time (i.e. atomically).

The definition of atomic nodes uses the following criterion: all the expressions present in such a node depend on the same set of inputs. This relies on a *sensitivity analysis* of programs. We say that a causality path exists between a node N_1 (resp. an input i) and a node N_2 if there is at least one situation where the execution of N_2 depends on the execution of N_1 (resp. on the occurrence of i). In that case, all the possible intermediate nodes are also executed.

Definition 5.3. Two nodes N_1 and N_2 are sensitively equivalent iff for each input i : there is a causality path from i to $N_1 \Leftrightarrow$ there is a causality path from i to N_2 .

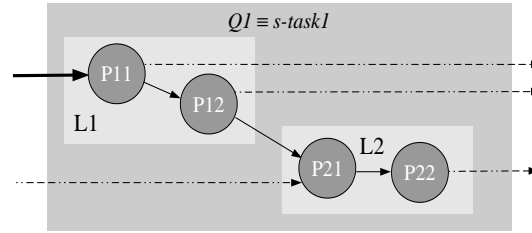


Fig. 16. Decomposition of an s-task into two clusters L_1 and L_2 .

Sensitively equivalent nodes belong to the same cluster. Inputs always precede outputs within a cluster. If a transformed program is endochronous, the resulting clusters are also endochronous. As a matter of fact, the clock hierarchy associated with each cluster is an endochronous sub-tree of the global clock tree characterizing the program. Hence, this ensures a deterministic execution of each cluster. FIG. 16 shows a decomposition of the s-task Q_1 into two clusters L_1 and L_2 . The input of the sub-program P_{11} (bold-faced arrow) is originally an input of P . The other arrows represent communications between s-tasks (These message exchanges are local to P). We can notice that after this second transformation, the semantic equivalence of the initial program and the resulting one is strictly preserved.

The two transformations presented above describe a partitioning of SIGNAL programs following a multi-task multi-processor architecture. The instantiation of such a description in the IMA model consists in using the ARINC component models we have described in Section 4 (APEX services, processes, partitions).

5.2 Instantiation of SIGNAL programs in the IMA model

We first present this instantiation at processor level then the approach could be generalized to the multi-processor level. From the above transformations, a processor can be considered as a graph where nodes are represented by clusters. Therefore, the partitioning of a given SIGNAL program following the IMA architecture model is obtained through the following steps :

—**Step 0: Distribute the program on the available processors.** Here, we assume a given distribution function. The program is transformed into s-tasks.

	$p - OS$	(partition-level OS)
	$cont_i$	(“control“ part of a process p_i)
	b_{ij}	(block)
p	$::= p - OS \mid p_1 \mid \dots \mid p_n$	(partition)
p_i	$::= cont_i \mid comp_i$	(process)
$comp_i$	$::= b_{i1} \mid \dots \mid b_{im_i}$	(“compute“ part of a process p_i)

Fig. 17. Modeling rules of IMA partitions.

In practice, this step is often an expert matter. However, there exist tools that can help to achieve this kind of task (e.g SYNDEX [Grandpierre and Sorel 2003]).

- Step 1: For each processor, transform the associated s-task into a graph of clusters.** This task is automatically performed by the SIGNAL compiler.
- Step 2: For each processor, associate clusters with partitions/processes.** The first decision about the graph of clusters resulting from the previous step consists in choosing a partitioning of clusters into IMA partitions/processes. In other words, we have to identify clusters that can be executed within the same partition/process. In our simple example, we decide to model the graph associated with $Q1$ (cf. FIG. 16) by one partition. Once partitions are chosen, the graph corresponding to each of them is decomposed into sub-graphs. These contain the clusters that should be executed by the same process. In the example, clusters associated with the “partition $Q1$ ” form the set of instruction blocks of a single process. The decomposition of the graph of clusters into partitions and processes relies on the following criterion: clusters that strongly depend on each other are first associated with the same process; then, the resulting processes (i.e. sub-sets of clusters), which strongly depend on each other are also put together in the same partition. This repartition has the advantage to greatly reduce inter-process communication costs. In the next step, the program can be effectively instantiated using our library of models (see Section 4).

—**Step 3: Instantiate the program in the IMA model.** Two phases are considered: first, we instantiate processes then partitions. An overview of the basic components used is given in FIG. 17. The symbol “|” denotes the synchronous composition. The following transformations are defined:

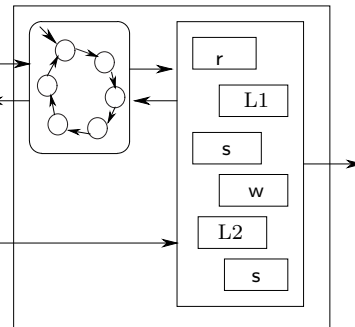
(1) Description of the process associated with a set of clusters:

- The definition of the CONTROL part of the process relies on dependencies between clusters. Clusters are executed sequentially with respect to these dependencies.
- Each cluster is “embedded” in a block within the COMPUTE part of the process.
- The internal communications between the clusters of a sub-graph associated with a process are modeled using local state variables (i.e. those defined by the *delay* construct). These variables enable to memorize exchanged data. On the other hand, communications between sub-graphs of clusters from different processes are modeled with APEX services. For each entry point (resp. exit point) of a sub-graph, a block containing a suitable communication or synchronization service call is added in the COMPUTE part of the associated process model. When the process becomes active, this block is executed just before (resp. after) the block that contains the cluster concerned by the entry point (resp. the exit point). The choice of the suitable service to call here depends on the desired type of communication. The program by itself is not enough to decide which service suits better. So, an expert knowledge is required. If the type of communication needs to store messages in a bounded queue, services associated with a buffer will be preferred to those related to a blackboard. On the contrary, if the communication only requires one memory-place, the blackboard will be more appropriate. Events are preferred to the two previous communication mechanisms when the exchanged data is specifically used to notify a process about certain condition. The services associated with semaphores are used for synchronization.

(2) Description of the partition associated with a set of clusters:

- The component corresponding to the partition-level OS (containing among other things the scheduler, which manages process execution within the partition) is added to the processes defined at the previous phase.
- The communication and synchronization mechanisms used by the APEX services added in the previous phase are created (e.g. for a *send_buffer* service call, a *buffer* should be created in which the messages can be stocked). This creation is done for example within the GLOBAL_OBJECTS sub-part of the partition, as illustrated in FIG. 13.

Example. On the right, we outline a process model resulting from the transformation of *Q1*. There are six blocks where two contains clusters *L1* and *L2*. The other blocks have been added for communication: *r*, *s* and *w* respectively denote a read request (receive_buffer or read_blackboard), an event notification (set_event), and an event notification waiting (wait_event). Of course, the way blocks are added is determined by the structure of *Q1*. But, the choice of the communication mechanisms (e.g. buffer or blackboard for the block *r*) contained in these blocks requires expert decisions. For instance, the process is supposed here to wait for some event notification between *L1* and *L2*. So, we use *w* instead of *r*, etc. The automaton described in the control part gives the execution order of the blocks with respect to the precedence constraints of the cluster graph. The corresponding partition is obtained by considering the phase 2 of **Step 3**, in the current section.



On each processor with multiple partitions, a model of a partition scheduler is required. Partition management is based on a time sharing strategy. Therefore, we have to compose a component (corresponding to the module-level OS - see Section 3) with partition models. A model of such a component is similar to the partition-level OS in that its definition relies on the use of APEX services, except that the scheduling policy differs.

To conclude this section, we briefly mention another study in which our ARINC concepts models have been used [Talpin et al. 2003]. We defined a technique in order to import resource constrained, multi-threaded, Real-Time Java programs, together with their runtime system API, into POLYCHRONY. We put this modeling technique to work by considering a formal, refinement-based, design methodology that allows for a correct by construction remapping of the initial threading architecture of a given Java program on either a single-threaded target or a distributed architecture. One advantage of the technique is the generation of stand-alone (JVM-less) executables and remapping of threads onto a given distributed architecture or a prescribed target real-time operating system. This allows for a complete separation of the virtual threading architecture of the system described in Java and its actual real-time and resource constrained implementation.

In Section 2, we mention that a great advantage of SIGNAL-based modeling is the possibility to formally analyze descriptions. Both functional and temporal properties can be addressed using different techniques. For instance, the compiler allows to check the absence of cyclic definitions in a program or the endochrony property of this program. These properties are necessary if one needs to generate the code associated with the program, e.g., for simulation. In [Gamatié and Gautier 2003a], we show how the SIGALI model-checker can be used to verify safety properties. In the next section, we rather concentrate on timing issues. We expose a high-level technique, which is entirely defined in SIGNAL that allows to approximate the execution time of modeled applications.

6. A MODULAR APPROACH FOR TIMING ANALYSIS

Timing issues can be addressed using the performance evaluation technique implemented in POLYCHRONY [Kountouris 1998]. The temporal analysis of the SATMAINT application presented in Section 6.2 is based on this technique. It relies on the principle introduced in Section 2.5, which consists in using an observer program, referred to as program *interpretation* in the sequel, to check properties of a given program. Section 6.1 presents how these interpretations are defined for any SIGNAL process. On the other hand, the timing analysis is considered on a lower level. It enables to check the latency of programs so as to validate the synchrony assumption. The main advantage is that we don't need to wait for the implementation phase for this validation.

6.1 General principle

An interpretation of a SIGNAL process is another process that exposes a different view of the initial one while preserving its control part. The temporal interpretation allows to check how an implementation of a specified application behaves over time. Let us consider a SIGNAL process P , its temporal interpretation for an implementation I is denoted by $T(P_I)$, where P_I is the SIGNAL process that models an implementation I of P . Thus, if an application specified by P may have different implementations $I(1)$ to $I(k)$, these implementations are modeled by $P_{I(i), i \in [1, k]}$, and for each $P_{I(i)}$ a temporal interpretation $T(P_{I(i)})$ can be derived. This way, a comparative performance evaluation of the implementations can be performed and their corresponding design space can be effectively explored before committing the design to one particular implementation. Such an approach permits to concentrate the design effort on a set of candidate implementations.

For a SIGNAL process P , we have:

$$P = C_P | D_P$$

where C_P and D_P are respectively the control and data parts of P . Similarly for an interpretation of P , we have:

$$T(P) = C_{T(P)} | D_{T(P)}$$

In particular, for the control part, we have:

$$C_{T(P)} = C_P.$$

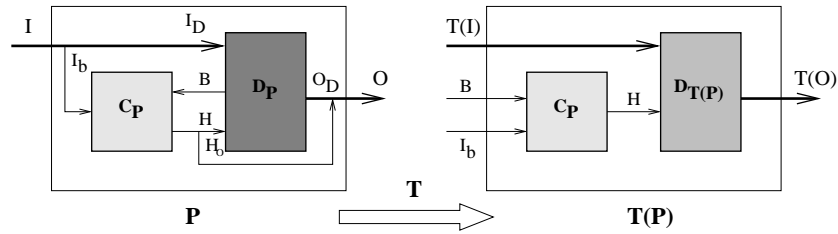


Fig. 18. Temporal interpretation of a SIGNAL process P .

The process of obtaining an interpretation $T(P)$ of a process P is graphically depicted in Figure 18. The data part D_P of the process P computes output values O_D from input values I . The computations are conditioned by activation events H computed in the control part C_P . To compute the activation conditions H , C_P uses boolean input signals I_b and intermediate boolean signals B computed by D_P . Finally, there are output events H_O computed by C_P . We can observe that the control parts of P and its interpretation $T(P)$ are identical while the data computation parts differ. In $T(P)$, data computations are devoted to temporal information whereas in P they strictly concern functional information.

Temporal interpretation of a simple process. As a general observation, a SIGNAL program is recursively composed of sub-processes, where elementary sub-processes are primitive constructs, called *atomic nodes*. A transformation of such a program substitutes each of its signals x with a new signal representing its timestamp (i.e. its availability dates) tsp_x , automatically replacing atomic nodes with their temporal model counter-part. The resulting time model is composed with the original functional description of the application (using the standard synchronous composition). Each signal x has the same clock as its associated date information tsp_x . The simulation of the resulting program reflects both the functional and timing aspects of the original program. Obviously, a less strict temporal model can be designed in order to perform faster simulation (or formal verification). It is

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

sufficient to consider more abstract representations either of the program or of its temporal model.

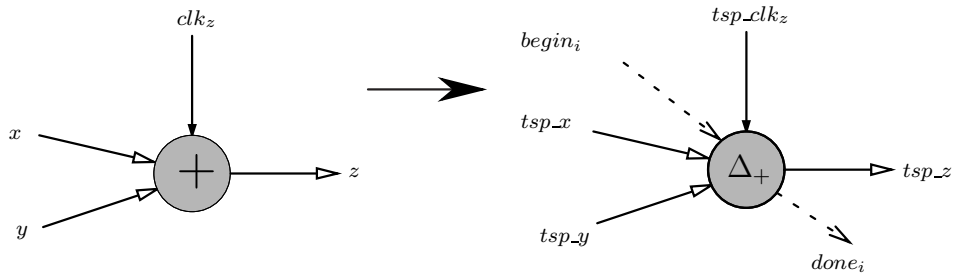


Fig. 19. Node associated with $z := x + y$ (left); and its temporal model (right).

The temporal interpretations of SIGNAL primitive constructs are collected in a *library of parameterized cost functions*. For a program to be interpreted, the library is extended with interpretations of external function calls and other separately compiled processes, which appear in the program. To illustrate the technique, we consider a simple SIGNAL program. For more complex examples, the reader may refer to [Kountouris 1998]. So, let us consider the following primitive construct: $z := x + y$. It is represented by the atomic node depicted by Figure 19, on the left hand side. Besides the input values x and y , this node also requires a clock information, denoted by the signal clk_z , which triggers the computation of the output value z . The associated temporal model is represented by the node illustrated on the right hand side. The SIGNAL program corresponding to this temporal model, called **T_Plus**, is depicted by Figure 20. In this model, **MAXn** denotes a SIGNAL process that returns the maximum value of n inputs among those that are present at a given instant (i.e., inputs are not constrained to be simultaneously present). The notations **type_x** and **type_y** represent the types of x and y respectively. The input signal **tsp_clk_z** is associated with the trigger clock clk_z . Signals **begin_i** and **done_i** have been added in order to express the end of execution of a given node so that the following nodes could be executed. The presence of **begin_i** in a node means that the preceding ones (following the scheduling order chosen for node execution), have already produced all their output timestamps. The presence of

`done_i` means that the current node has calculated all its output timestamps (i.e., `done_i` becomes `begin_i+1`). The timestamp of `z`, denoted by `tsp_z`, is the sum of the maximum timestamp of inputs and the delay of the addition operation, some Δ_+ . This quantity Δ_+ depends on the desired implementation, on a specific platform. It has to be provided in some way by the user, with respect to the considered platform. In the current implementation in POLYCHRONY, the value Δ_+ is provided by a function `DELTA_ADD` which has the types of the operands as parameters and which fetches the required value from some table. Following the same idea, each primitive construct of SIGNAL has been associated with its temporal interpretation. As a result of the compositionality of SIGNAL specifications, the same principle can be applied at any level of granularity.

```

process T_Plus{ type_x, type_y; }
  ( ? tsp_type tsp_x, tsp_y, tsp_clk_z, begin_i;
    ! tsp_type tsp_z, done_i; )
  (| tsp_z := MAX2( MAX3( tsp_x, tsp_y, tsp_clk_z),
                    begin_i when ^tsp_z) + DELTA_ADD{type_x, type_y}()
    | done_i := (tsp_z default begin_i) cell ^done_i
  |);

```

Fig. 20. A temporal interpretation of `z := x + y` in SIGNAL.

In addition to the library of cost functions of primitive constructs, the implementation also requires *platform-dependent information* (e.g. the delay of the addition of two integer numbers on a given processor). For instance, the sum of integer numbers coded on 32 bits will take one cycle on a processor with a 32-bit adder (this is the case of the Intel Pentium IV processor). On the other hand, the same operation requires more than one cycle on a processor with only a 16-bit adder (like the Intel 8086 processor). In the example exposed in Figure 20, this information is

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

obtained via the DELTA_ADD call.

In the following, we are interested in determining an approximation of the execution time of a real world application specified in SIGNAL.

6.2 Application to a real world case study

Let us consider an application, called SATMAINT, represented by a single partition model as it has been shown in [Gamatié et al. 2004]. FIG. 21 shows a *co-simulation* of this application together with its temporal interpretation T_SATMAINT (the prefix notation “T_” stands for temporal).

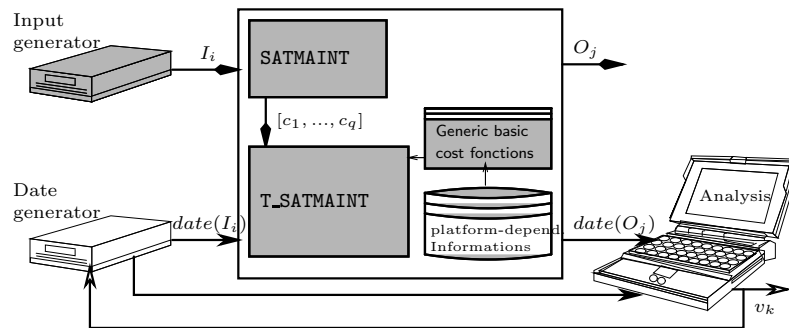


Fig. 21. Co-simulation of SATMAINT with its temporal interpretation.

At each simulation step, the timestamp of outputs $tsp(O_j)$ depends on the timestamp of inputs $tsp(I_i)$ and on the *control configuration*, represented by a “valuation” of boolean signals vector $[c_1, \dots, c_q]$ computed in the original program (represented by B in FIG. 18). The boolean c_i are mainly signals that are defined by numerical expressions with comparison, for instance, $c1$ in the primitive construct $c1 := x > 5$. Typically, $c1$ may be used as control information in the program.

In fact, the control parts of SATMAINT and T_SATMAINT are identical, only data parts differ: in SATMAINT, the data part computes functional results while in T_SATMAINT, it yields timestamp informations. The vector $[c_1, \dots, c_q]$ contains intermediate boolean signals evaluated by the data part of SATMAINT that are

needed by T_SATMAINT to compute output information. Note that in a straightforward approach, it is possible to provide a set of vectors that covers all the possible combinations for the control flow. A better way is to take into account the existing relationships between these booleans such as provided by the clock calculus of SIGNAL (this is expressed through the composition of the original program and its temporal interpretation).

In practice, we mainly raise one difficulty about the implementation of the schema illustrated in Figure 21. It is due to the scalability issue which can become problematic when compiling large application programs. For example, the program resulting from the composition of the application model together with its temporal interpretation - (| SATMAINT | T_SATMAINT |) - can be huge and may not facilitate the compilation. This issue is widely addressed in [Amagbegnon et al. 1994] and [Amagbegnon et al. 1995] where authors present several experiments on the analysis of typical SIGNAL application programs using the compiler. The results are given in different tables that show the time and memory limits depending on the size of each studied application. The solution we adopt here consists in using a modular evaluation approach, which is exposed in the sequel.

The modularity of the SIGNAL language allows to construct a program from other programs by composition. Therefore, this principle also applies to the construction of the temporal interpretation of the partition SATMAINT, which is quite large. For that, we consider a splitting of the corresponding SIGNAL model into subparts with reasonable size (e.g. such a subpart could be an IMA process). This makes them easier to address. For each subpart, we therefore define an associated temporal interpretation. Afterwards, the resulting model can be composed with the concerned subpart. The program obtained from this composition is abstracted in order to take into account only information that is relevant for the considered observation (abstractions also contribute to reduce the size of a program). Finally, the global model that consists of the composition of the application with its temporal interpretation is obtained by composing abstracted subprograms.

Let $P \equiv P_1 \mid \dots \mid P_n$ denote the program corresponding to the application considered for temporal analysis. We want to define the program $P' \equiv (\mid P \mid T_P \mid)$ that is used for the analysis, where T_P is the temporal model of P . The same program can be also rewritten as: $P' \equiv P'_1 \mid \dots \mid P'_n$. Each P'_i denotes the composition of a subprogram P_i of P , with its associated temporal interpretation T_{P_i} . The following steps are identified in order to carry out experiments:

- (1) *Partial definition of temporal interpretations*: for each sub-program $P_{i,i \in \{1, \dots, n\}}$ of P , we define the corresponding temporal model T_{P_i} .
- (2) *Composition of each subpart of the application with its associated temporal interpretation, then abstraction*: this step defines the subprograms P'_i ($i \in \{1, \dots, n\}$), which constitute the simulation program P' that we want to construct. The abstraction aims to keep only relevant information of these subprograms for the co-simulation. It follows that $P'_i \equiv \alpha(P_i \mid T_{P_i})$, where α denotes the abstraction (e.g. a program can be abstracted by considering only its control part—boolean and synchronization signals—or by approximating the value of numerical signals—for instance, by dealing with domains of intervals instead of point-wise domains).
- (3) *Construction of the global model for simulation*: this model results from the composition of subprograms P'_i , defined at the previous step, i.e.

$$P' \equiv P'_1 \mid \dots \mid P'_n.$$

On the other hand, we notice that the above method can be applied in a recursive way. Still considering the above program P , if the size of subprograms P_i ($i \in \{1, \dots, n\}$) is important, we can also use the same method for each one in order to define the corresponding P'_i ($i \in \{1, \dots, n\}$). The global simulation model then results from the composition of P'_i s.

To apply the method to the partition SATMAINT, we first decomposed it into subparts, represented by its processes (it contains eight processes). For instance, its process identified as `PROC_8` is composed of eight blocks. We begin by defining

the interpretation of the COMPUTE subpart of this process:

$$T_COMPUTE \equiv T_BLOCK_0 \mid \dots \mid T_BLOCK_7.$$

In this composition, we suppose that the temporal model of each block is obtained without necessarily having to consider its decomposition as is done for subprograms of a larger size. The temporal interpretation of the CONTROL subpart of the process is determined in a similar way. The composition of both temporal models gives the model for PROC_8:

$$T_PROC_8 \equiv T_COMPUTE_8 \mid T_CONTROL_8.$$

We proceed in the same way for the other subparts of the partition. Then, we compose the resulting temporal model with its associated program subpart. For PROC_8, it follows:

$$(| \text{PROC}_8 \mid T_PROC_8 |).$$

This process can now be simplified by considering approximations (or abstractions). This facilitates the compilation of the global program. It is important to consider approximations that do not affect the computed timing information by increasing the set of possible behaviors of a program. In particular, a good approximation must preserve the control part of a program. Given a SIGNAL program P, the compiler is able to generate such an approximation in which the only specified information are clock relations and dependencies between the signals of P. Hence, the resulting program could be considered for co-simulation with the temporal interpretation of P.

On the other hand, we can consider approximations of the temporal interpretation of a program. Here also, the control part must be preserved. Only the data part needs to be redefined in order to be reduced. For instance, let us consider a subpart of the application that performs a complex operation, which requires a constant duration δ on a target platform (by “complex”, we mean an operation that requires several elementary operations, e.g. product of two matrices). The temporal model of such a subpart can be defined in a simple way by adding the

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

constant δ to the timestamps corresponding to inputs availability in order to compute timestamps associated with outputs. This interpretation is simpler than the one obtained by composing the temporal models of all intermediate operations that are carried out by the considered subprogram. We can also mention other simplifications, which consist in considering worst case execution times (WCET) [Puschner and Burns 2000] in the definition of the temporal interpretation of a sub-program. Over the recent years, there have been several promising studies that address major issues in WCET computation such as branch prediction [Bodin and Puaut 2005] or caching [Puaut and Decotigny 2002]. Finally, instead of considering a single WCET value for approximations, one may use a range of possible execution times. This last possibility provides more accurate information since the order of execution of statements reflected by the control part is taken into account.

Global observations. Modularity and abstraction play a central role for scalability in our design approach. Basically, the description of a large application is achieved by specifying first, either completely or partially (by using abstractions), sub-parts of the application. After that, the resulting programs can be composed in order to obtain new components. These components can be also composed and so on, until the application description is complete. The construction of the global simulation model of SATMAINT for temporal issues relies on the same principle as the description of the application itself. A crucial issue about the design of safety critical systems, such as avionics, is the correctness of these systems. In POLYCHRONY, the functional properties of a system can be checked using tools like the compiler or the model checker SIGALI. Here, we address temporal aspects of programs. For that, we used a technique consisting in co-simulating the program under analysis with an associated observer (also referred to as temporal interpretation) defined in SIGNAL. The observer is another program which has the same control as the observed one, but its data part reflects the temporal dimension of the analyzed program. Using SIGNAL for both the model of an application and its associated temporal interpretation results in unified descriptions upon which available tools and techniques remain applicable.

7. RELATED WORK

Over the past decade, several approaches have been proposed for the design of embedded real-time systems. We first mention typical ones from different families of approaches that are related to our work. Then, we present a few studies dedicated to avionic systems.

7.1 On the design of embedded real-time application in general

TAXYS [Closse et al. 2001] has been proposed for the design and validation of real-time embedded applications. It is partly based on the synchronous approach by using the synchronous language ESTEREL with an associated compiler. The model checker KRONOS is used for timing and schedulability analysis. TAXYS combines the synchronous approach and timed automata theory.

The GIOTTO approach [Henzinger et al. 2001] considers a specific language for the design that is used to define abstract models of embedded control systems. The language has a time-trigger semantics that facilitates time predictability for system analysis. The associated compiler and a runtime library enable the implementation on a target platform. One interesting common characteristic of the GIOTTO formalism and the synchronous ones is that they both allow platform-independent specifications.

Component-based design techniques have revealed advantages, re-usability being the most prominent. We mention a few approaches such as METAH [Vestal 1997], VEST [Stankovic et al. 2003] and CADENA [Hatcliff et al. 2003]. In the former, the design activity relies on the architecture description language *MetaH*. The approach addresses embedded real-time, distributed avionic applications. It proposes a tool-set for the description and combination of software and hardware components to construct a system. Facilities are also provided for real-time analysis. The VEST approach is similar with METAH. It also provides a tool that supports infrastructure creation, embedded system composition, and mapping of passive software

ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

components to active run-time structures (such as tasks). A set of analysis tools is also available for real-time and reliability analysis. To enable this, a library of micro-components, components, and infrastructures has been defined. Micro-components are passive software such as interrupt handlers, dispatchers, and plug and unplug primitives. The tool supports dependency checks and composition. However, VEST does not support formal proof of correctness. Finally, CADENA offers a framework for CORBA component model development, which allows a user to explore how static analysis and model-checking can be integrated. In [Hatcliff et al. 2003], an illustration is given of its application to simple avionics systems based on the Boeing Bold Stroke architecture [Sharp and Roll 2003].

Other approaches consider a separation of concerns in system design. The SYNDEX approach [Grandpierre and Sorel 2003] separates aspects inherent to an application from those related to the implementation platform. SYNDEX proposes a methodology called *algorithm architecture adequation* (AAA) supported by system-level Computer-Aided Development software called for the design of distributed embedded real-time systems. The AAA methodology covers the whole design steps: from the specification of functionalities (i.e. the software level) to the deployment on target multi-processor architectures including specific integrated circuits (i.e. the hardware level). This is achieved using a graphical environment, which allows the designer to manually and/or automatically identify efficient ways to distribute application functionalities on the target architecture (the criterion is to satisfy timing requirements and to minimize hardware resources). The distribution relies on graph representations, and takes non-functional requirements inputs (e.g. real-time performances). Finally, a possibility of automatically generating code corresponding to the specified system is offered. In addition to SYNDEX, there is the AUTOFOCUS approach [Romberg 2002], which follows similar ideas. It addresses distributed system design by offering integrated hierarchical description techniques for different views of a systems: the structure of a system (its components and communication between them), its behavioral description and the way the system interacts with its environment.

Among approaches dedicated to the design of embedded real-time systems, only a few of them address both continuous and discrete activities of such systems. The approach based on the CHARON language [Alur et al. 2003] enables modular specifications of interacting *hybrid systems* based on the notions of agent (a building block) and mode (an hierarchical state machine). The description of a system follows a two level hierarchy: the architectural level provides the structure of the system in terms of composed agents whereas the behavioral level indicates the interaction modes.

Finally, we mention the PTOLEMY approach [Lee 2001], which supports the modeling, simulation, and design of embedded systems. It integrates several computation models (e.g. synchronous/reactive systems, continuous time, etc.) in order to deal with concurrency and time. A key point is that the designer can simulate different kinds of interactions between active components. Therefore, the focus is mainly on the choice of suitable computation models for an appropriate type of behavior in the system.

7.2 On the design of avionic applications

We mention a few studies addressing the design of embedded real-time systems in the avionic domain. The main objective of the COTRE approach [Berthomieu et al. 2003] consists in providing the designer with a methodology, an Architecture Description Language (ADL) called *Cotre*, and an environment to describe, verify and implement embedded avionic systems. The Cotre language distinguishes two different views for descriptions: a user view expressed using the *Cotre for User* language (termed *U-Cotre*) and a view for verification (termed *V-Cotre*). In fact, the latter plays the role of an intermediate language between U-Cotre and certain existing verification formalisms (e.g. timed automata, timed Petri nets). The authors argue that the use of formal techniques is one of the main differences between the Cotre language and other ADLS. COTRE is closely related to the approach based on the *Avionics Architecture Description Language* (AADL), which is developed by the International Society of Automotive Engineers (SAE) [AADL Coordination Com- ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

mittee 2002]. It is dedicated to the design of the software and hardware components of an avionic system and the interfaces between those components. The AADL definition is based on METAH (an ADL developed by Honeywell) [Vestal 1997]. It permits the description of the structure of an embedded system as an assembly of software and hardware components in a similar way. The AADL draft standard also includes a UML profile of the AADL. This enables the access to formal analysis and code generation tools through UML graphical specifications. While these approaches combine various formalisms and tools for the design of embedded real-time systems, our approach relies on the single semantic model of the SIGNAL language. It is very important to have a common framework in order to guarantee the correctness of the designs.

SCADE is one of the most established formal design environments in avionics domain [Camus and Dion 2003]. It is also based on the synchronous technology. It supports *correct by construction* methodology and automated generation of qualifiable implementation (level A with respect to DO-178B guidelines) from a high-level formal model of embedded applications. A main advantage is drastic savings in the development and validation efforts. Among important avionic projects in which the SCADE suite has been used, we can mention the Airbus A380 and Eurocopter.

Among specific studies related to IMA, we mention those concerning the two-level hierarchical scheduling aspects within IMA systems. In [Lee et al. 2000] Lee *et al.* present algorithms that provide cyclic partition and channel schedules for IMA systems. In the same context, Audsley and Wellings analyze response times of IMA applications [Audsley and Wellings 1996]. They point out the possibility of a large amount of jitter and discuss possible ways to reduce the jitter by checking process periods with respect to partitions periods. One of the techniques illustrated in this article for temporal analysis provides information on execution times of partitions. Thus, this information can be used when taking decisions in processor allocation to partitions. Further expected benefits of defining our approach in a formal framework are the available techniques and tools that help to address

some critical issues of IMA such as the partitioning, which still need to be further explored by researchers. Indeed, in current industrial practices, avionic functions with high critical level are designed using federated architectures (for instance, this is the case for the future Airbus A380). This is likely due to the fact that partitioning raises several questions that are not sufficiently addressed yet. Among these questions, we can mention the correctness of a partitioning, which is crucial. A formal description of partitioning requirements is proposed by Di Vito [Di Vito 1999], using the language of PVS (Prototype Verification System). However, this description only concerns space partitioning (time partitioning is not addressed). The use of SIGNAL enables to define a correct-by-construction partitioning, based on the *sensitivity analysis* [Sacres Consortium 1997]. Being able to guarantee the correctness of a given partitioning helps to reduce IMA certification efforts. A study addressing this last issue has been done by Conmy and McDermid [Conmy and McDermid 2001], who propose a high level failure analysis of IMA. The analysis is part of an overall IMA certification strategy. Finally, a presentation of the IMA-based communication network designed for the future Airbus A380 is given by Sánchez-Puebla and Carretero in [Sánchez-Puebla and Carretero 2003].

8. CONCLUSIONS

In this paper, we argue that the polychronous framework favors reliable designs of embedded real-time systems. The central idea is the definition of a library of polychronous models of architecture components. As in METAH [Vestal 1997] or VEST [Stankovic et al. 2003], this library includes active run-time components such as processes and functionalities of a real-time operating system.

We advocate a seamless design methodology including high level specifications using the modularity and re-usability features of the polychronous language SIGNAL, formal verification and performance evaluation, and automatic code generation. In such a context, the formal basis of SIGNAL is a key aspect for validation. This is essential to the design of safety critical systems.

Beyond the formal framework promoted by the present work for the design activity, it suggests a possible way to address the crucial issue of partitioning in integrated modular avionics architectures. Most modern aircraft still massively adopt the federated approach instead of IMA for highly critical functions. It seems that one main difficulty of that arises from the partitioning itself in such systems. We believe that the ideas exposed in this paper help to overcome this difficulty. For instance, the sensitivity analysis we use in our approach allows us to easily identify dependencies between sub-parts of a system, in a correct way. In this sense, the SYNDEX approach [Grandpierre and Sorel 2003] can be also mentioned here. In SYNDEX, the partitioning of an application relies on the performance of the implementation platform. This can be combined with our sensitivity analysis within a new version of our methodology where the distribution function previously assumed at **Step 0** (see Section 5) is now replaced by a strategy based on quantitative criteria. We then observe a complementarity of SYNDEX and our approach.

From a scientific point of view, the main contribution of this paper concerns the modeling of (bounded) asynchronous mechanisms using the synchronous approach. There have been many studies on this topic [Berry and Sentovich 1998] [Caspi 2001] [Halbwachs and Baghdadi 2002] [Gamatié and Gautier 2003a]. They turn out to promote *globally asynchronous locally synchronous* architectures (GALS) where a system is composed of sub-systems/components that execute synchronously and communicate asynchronously. In the current study, we presented how a typical asynchronous architecture and its associated mechanisms can be modeled using SIGNAL for validation.

From a methodological point of view, various design approaches have been combined within a unique general framework. We first considered a component-based approach in order to define a library of models. Then, we used these models to derive from a given SIGNAL description of an application, its corresponding IMA model by refining the initial description. We also show how temporal issues or the resulting description can be addressed in a modular way.

Finally, from an implementation point of view, this work leads to the definition of synchronous models of APEX-ARINC 653 services. They are freely available together within the POLYCHRONY platform at [ESPRESSO-IRISA.]. We faced the scalability question of the proposed approach by trying it out for a real world application.

ACKNOWLEDGMENTS

We wish to thank David Berner for his valuable comments on previous versions of this article.

REFERENCES

- AADL COORDINATION COMMITTEE. 2002. Avionics architecture description language. In *AADL seminar*. Society of Automotive Engineers, Toulouse, France.
- AIRLINES ELECTRONIC ENGINEERING COMMITTEE. 1997a. ARINC report 651-1: Design guidance for integrated modular avionics. Tech. Rep. 651, Aeronautical radio, Inc., Annapolis, Maryland. November.
- AIRLINES ELECTRONIC ENGINEERING COMMITTEE. 1997b. ARINC specification 653: Avionics application software standard interface. Tech. Rep. 653, Aeronautical radio, Inc., Annapolis, Maryland. January.
- ALUR, R., DANG, T., ESPOSITO, J., HUR, Y., IVANCIC, F., KUMAR, V., LEE, I., MISHRA, P., PAPPAS, G., AND SOKOLSKY, O. 2003. Hierarchical modeling and analysis of embedded systems. *IEEE Press 91*, 1, 11–28.
- AMAGBEGNON, T., BESNARD, L., AND LE GUERNIC, P. 1994. Arborescent canonical form of boolean expressions. Tech. Rep. 2290, INRIA. June. Available at <http://www.inria.fr/rrrt/rr-2290.html>.
- AMAGBEGNON, T., BESNARD, L., AND LE GUERNIC, P. 1995. Implementation of the data-flow synchronous language SIGNAL. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'95)*. 163 – 173.
- ARVIND AND GOSTELOW, K. 1978. *Some Relationships between Asynchronous Interpreters of a Dataflow Language*. North-Holland.
- AUDSLEY, N. AND WELLINGS, A. 1996. Analysing APEX Applications. In *Proceedings of Real Time Systems Symposium (RTSS'96)*. IEEE Press, Washington, DC, USA.
- BENVENISTE, A. 1998. Safety critical embedded systems: the SACRES approach. In *Proceedings ACM Transactions on Software Engineering and Methodology*, Vol. V, No. N, November 2005.

of *Formal techniques in Real-Time and Fault Tolerant Systems (FTRTFT'98 school)*, Lyngby, Denmark.

- BENVENISTE, A., CASPI, P., EDWARDS, S., HALBWACHS, N., LE GUERNIC, P., AND DE SIMONE, R. 2003. The synchronous languages twelve years later. *Proceedings of the IEEE 91*, 1 (January), 64–83.
- BENVENISTE, A., CASPI, P., LE GUERNIC, P., MARCHAND, H., TALPIN, J., AND TRIPAKIS, S. 2002. A protocol for loosely time-triggered architectures. In *Proc. of 2002 Conference on Embedded Software, EMSOFT'02*, J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS vol 2491, Springer Verlag.
- BENVENISTE, A. AND LE GUERNIC, P. 1990. Hybrid dynamical systems theory and the SIGNAL language. *IEEE transactions on Automatic Control 35*, 5 (May), 535–546.
- BERRY, G. AND SENTOVICH, E. 1998. Embedding synchronous circuits in GALS-based systems. In *Proceedings of the Sophia-Antipolis conference on Micro-Electronics (SAME'98)*. Sophia-Antipolis, France.
- BERTHOMIEU, B., RIBET, P.-O., VERNADAT, F., BERNARTT, J. L., FARINES, J.-M., BODEVEIX, J.-P., FILALI, M., PADIOU, G., MICHEL, P., FARAIL, P., GAUFFILET, P., DISSAUX, P., AND LAMBERT, J.-L. 2003. Towards the verification of real-time systems in avionics: the cotre approach. *Electronic Notes in Theoretical Computer Science 80*, 1–16.
- BODIN, F. AND PUAUT, I. 2005. A WCET-oriented static branch prediction scheme for real-time systems. In *Proc. of the 17th Euromicro Conference on Real-Time Systems*. Palma de Mallorca, Spain.
- CAMUS, J.-L. AND DION, B. 2003. Efficient development of airborne software with SCADE suite. In *Esterel Technologies white paper*. <http://www.esterel-technologies.com>.
- CASPI, P. 2001. Embedded control: From asynchrony to synchrony and back. In *Proceedings of the first International Workshop on Embedded Software (EMSOFT'01)*. Th. A. Henzinger and Ch. M. Kirsch, Eds., LNCS 2211, Springer Verlag, Lake Tahoe.
- CLOSSE, E., POIZE, M., PULOU, J., SIFAKIS, J., VENIER, P., WEIL, D., AND YOVINE, S. 2001. Taxys: a tool for the development and verification of real-time embedded systems. In *Proceedings of Computer Aided Verification, CAV'01*. Lecture Notes in Computer Science 2102, Springer-Verlag, Paris, France.
- CONMY, P. AND McDERMID, J. 2001. High Level Failure Analysis for Integrated Modular Avionics. In *Proceedings of the 6th Australian Workshop on Industrial Experience with Safety Critical Systems and Software*. Brisbane, Australia.
- DAWS, C. AND YOVINE, S. 1995. Two Examples of Verification of Multirate Timed Automata with KRONOS. In *Proceedings of the 16th IEEE Real Time Systems Symposium (RTSS'95)*. IEEE Press, Pisa, Italy.
- ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

- DENNIS, J. B., FOSSEN, J. B., AND LINDERMAN, J. P. 1974. Data flow schemas. In *International Symposium on Theoretical Programming*, A. Ershov and V. A. Nepomniaschy, Eds. Lecture Notes in Computer Science, 5, Springer-Verlag, 187–216.
- DI VITO, B. 1999. A Model of Cooperative Noninterference for Integrated Modular Avionics. In *Proceedings of Dependable Computing for Critical Applications (DCCA-7)*. San Jose, CA.
- ESPRESSO-IRISA. The POLYCHRONY website. <http://www.irisa.fr/espresso/Polychrony>.
- GAMATIÉ, A. AND GAUTIER, T. 2002. Synchronous modeling of modular avionics architectures using the SIGNAL language. Tech. Rep. 4678, INRIA. December. Available at <http://www.inria.fr/rrrt/rr-4678.html>.
- GAMATIÉ, A. AND GAUTIER, T. 2003a. The SIGNAL approach to the design of system architectures. In *10th International Conference and Workshop on the Engineering of Computer-based Systems, Huntsville - Alabama*.
- GAMATIÉ, A. AND GAUTIER, T. 2003b. Synchronous modeling of avionics applications using the SIGNAL language. In *Proceedings of the 9th IEEE Real-time/Embedded technology and Applications symposium (RTAS'03)*. IEEE Press, Washington D.C., USA.
- GAMATIÉ, A., GAUTIER, T., AND LE GUERNIC, P. 2004. An example of synchronous design of embedded real-time systems based on IMA. In *Proceedings of the 10th International Conference on Real-time and Embedded Computing Systems and Applications (RTCSA'04)*. LNCS, Springer Verlag, Gothenburg, Sweden.
- GAUTIER, T. AND LE GUERNIC, P. 1999. Code generation in the SACRES project. In *Safety-critical Systems Symposium, SSS'99*, Springer. Huntingdon, UK.
- GOSHEN-MESKIN, D., GAFNI, V., AND WINOKUR, M. 2001. SAFEAIR: An integrated development environment and methodology. In *Proceedings of the INCOSE 2001*. Melbourne.
- GRANDPIERRE, T. AND SOREL, Y. 2003. From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Formal Methods and Models for Codesign Conference*. Mont Saint-Michel, France.
- HALBWACHS, N. AND BAGHDADI, S. 2002. Synchronous modelling of asynchronous systems. In *Conference on Embedded Software (EMSOFT'02)*. J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS 2491, Springer Verlag, Grenoble, France, 240–251.
- HALBWACHS, N., LAGNIER, F., AND RAYMOND, P. 1993. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology*. Springer Verlag 1994, ISBN 3-540-19852-0, Enschede, The Netherlands, 83–96.
- HATCLIFF, J., DENG, W., DWYER, M., JUNG, G., AND PRASAD RANGANATH, V. 2003. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 2003 International Conference on Software Engineering (ICSE 2003)*, Portland, Oregon.
- ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

- HENZINGER, T., HOROWITZ, B., AND KIRSCH, C. 2001. Embedded control systems development with giotto. In *Proceedings of LCTES*. ACM SIGPLAN Notices.
- KAHN, G. 1974. The semantics of a simple language for parallel programming. In *Information Processing 74*, J. L. Rosenfeld, Ed. North-Holland, 471–475.
- KOUNTOURIS, A. 1998. Outils pour la validation temporelle et l’optimisation de programmes synchrones. Ph.D. thesis, Université de Rennes I, Rennes, France.
- KOUNTOURIS, A. AND LE GUERNIC, P. 1996. Profiling of SIGNAL programs and its application in the timing evaluation of design implementations. In *Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems*. HP Labs, Bristol, UK, 6/1–6/9.
- LE GUERNIC, P., TALPIN, J.-P., AND LE LANN, J.-C. 2003. Polychrony for System Design. *Journal for Circuits, Systems and Computers* 12, 3 (April), 261–304.
- LEE, E. 2000. What’s ahead for embedded software? *IEEE Computer Magazine* 33, 9 (September), 18–26.
- LEE, E. 2001. Overview of the Ptolemy project. Tech. Rep. UBC/ERL M01/11, University of California, Berkeley. March.
- LEE, Y.-H., KIM, D., YOUNIS, M., ZHOU, J., AND MCELROY, J. 2000. Resource Scheduling in Dependable Integrated Modular Avionics. In *Proceedings of the International Conference on Dependable Systems and Networks*.
- MARCHAND, H., BOURNAI, P., LE BORGNE, M., AND LE GUERNIC, P. 2000. Synthesis of discrete-event controllers based on the SIGNAL environment. *Discrete Event Dynamic System: Theory and Applications* 10, 4 (October), 325–346.
- PNUELI, A. 2002. Embedded Systems: Challenges in Specification and Verification. In *Proceedings of the second International Conference on Embedded Software (EMSOFT 2002)*. J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS 2491, Springer Verlag, Grenoble, France, 252–265.
- PUAUT, I. AND DECOTIGNY, D. 2002. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proc. of the 23rd IEEE International Real-Time Systems Symposium*. Austin, TX, USA.
- PUSCHNER, P. AND BURNS, A. 2000. A review of worst-case execution-time analysis. *Journal of Real-Time Systems* 18, 2/3 (May), 115–128.
- ROMBERG, J. 2002. Model-based deployment with autofocus: a first cut. In *14th Euromicro Conference on Real Time Systems (ECRTS’02), Work In Progress session*. IEEE Press, Vienna, Austria, 41–44.
- RUSHBY, J. 1999. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Tech. Rep. CR-1999-209347, NASA Langley Research Center. June. Available at <http://www.csl.sri.com/users/rushby/partitioning.html>.
- SACRES CONSORTIUM. 1997. The semantic foundations of SACRES. Tech. Rep. EP 20897. March. ACM Transactions on Software Engineering and Methodology, Vol. V, No. N, November 2005.

- SÁNCHEZ-PUEBLA, M. AND CARRETERO, J. 2003. A new Approach for Distributed Computing in Avionics Systems. In *Proceedings of the 1st International Symposium on Information and Communication Technologies*. Dublin, Ireland.
- SHARP, D. AND ROLL, W. 2003. Model-based integration of reusable component-based avionics systems. In *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS'2003*.
- SIFAKIS, J. 2001. Modeling real-time systems - challenges and work directions. In *Proceedings of the first International Workshop on Embedded Software (EMSOFT 2001)*. LNCS 2211, Springer Verlag, Tahoe City.
- STANKOVIC, J., ZHU, R., POORNALINGAM, R., LU, C., YU, Z., HUMPHREY, M., AND ELLIS, B. 2003. VEST: An aspect-based composition tool for real-time systems. In *Proceedings of the 9th IEEE Real-time/Embedded technology and Applications symposium (RTAS'03)*. IEEE Press, Washington D.C., USA.
- TALPIN, J.-P., GAMATIÉ, A., BERNER, D., LE DEZ, B., AND LE GUERNIC, P. 2003. Hard real-time implementation of embedded systems in java. In *International Workshop on Scientific Engineering of Distributed JAVA Applications*. Lectures Notes in Computer Science, Springer Verlag, Luxembourg.
- VESTAL, S. 1997. MetaH support for real-time multi-processor avionics. In *IEEE Workshop on Parallel and Distributed Real-Time Systems*. IEEE Press, Geneva, Switzerland.
- WADGE, W. W. 1979. An extensional treatment of dataflow deadlock. In *Semantics of Concurrent Computation*, G. Kahn, Ed. Lecture Notes in Computer Science, 70, Springer-Verlag, 285–299.
- WIRTH, N. 2001. Embedded systems and real-time programming. In *Proceedings of the first International Workshop on Embedded Software (EMSOFT 2001)*. Th. A. Henzinger and Ch. M. Kirsch, Eds., LNCS 2211, Springer Verlag, Tahoe City, California, 486–492.