Polychronous Design of Embedded Real-Time Applications

ABDOULAYE GAMATIÉ
INRIA Futurs, France
and
THIERRY GAUTIER, PAUL LE GUERNIC, and JEAN-PIERRE TALPIN
IRISA/INRIA, France

Embedded real-time systems consist of hardware and software that controls the behavior of a device or plant. They are ubiquitous in today's technological landscape and found in domains such as telecommunications, nuclear power, avionics, and medical technology. These systems are difficult to design and build because they must satisfy both functional and timing requirements to work correctly in their intended environment. Furthermore, embedded systems are often critical systems, where failure can lead to loss of life, loss of mission, or serious financial consequences. Because of the difficulty in creating these systems and the consequences of failure, they require rigorous and reliable design approaches. The synchronous approach is one possible answer to this demand. Its mathematical basis provides formal concepts that favor the trusted design of embedded real-time systems. The multiclock or *polychronous* model stands out from other synchronous specification models by its capability to enable 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 enable modular development of increasingly complex modern systems. The expressiveness of its underlying semantics allows dealing with several issues of 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 [**Software Engineering**]: Requirements/Specifications—*Methodologies (e.g., object-oriented, structured)* ; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; D.2.4 [**Software Engineering**]: Software Program/Verification—*Formal methods, validation*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

This study has been partially supported by the European project IST SafeAir (Advanced Design Tools for Aircraft Systems and Airborne Software—www.safeair2.org/safeair).
Authors' addresses: A. Gamatié, INRIA, Synergie Park, 6bis Ave. Pierre et Marie Curie 59260 Lezennes, France; email: gamatie@lifl.fr; T. Gautier, P. Le Guernic, J.-P. Talpin, IRISA/INRIA Campus Universitaire de Beaulieu, Avenue du General Leclerc, F-35042 RENNES Cedex, France; email: {gautier,paul.le_guernic,talpin}@irisa.fr.

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: Synchronous approach, Avionics, SIGNAL, IMA

ACM Reference Format:

Gamatié, A., Gautier, T., Le Guernic, P., and Talpin, J.-P. 2007. Polychronous design of embedded real-time applications. ACM Trans. Softw. Eng. Methodol. 16, 2, Article 9 (Apr. 2007), 44 pages. DOI = 10.1145/1217295.1217298 http://doi.acm.org/10.1145/1217295.1217298

1. INTRODUCTION

Embedded real-time systems consist of hardware and software that controls the behavior of a device or plant. They are ubiquitous in today's technological landscape and found in domains such as telecommunications, nuclear power, avionics, and medical technology. These systems are difficult to design and build because they must satisfy both functional and timing requirements to work correctly in their intended environment. Furthermore, embedded systems are often critical systems, where failure can lead to loss of life, loss of mission, or serious financial consequences. Because of the difficulty in creating these systems and the consequences of failure, they require rigorous and reliable design approaches. Over the past decade, high-level design of such systems has gained prominence in the face of rising technological complexity, increasing performance requirements, and tightening time-to-market constraints. Broad discussions of challenges in the design of these systems can be found in the 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 meet these requirements [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, in which the system execution is 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 multiclock, or *polychronous*, model [Le Guernic et al. 2003] stands out from other synchronous specification models by its capability to enable 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 enable modular development of increasingly complex modern systems. Using this approach, the design of each component can be addressed separately. The expressiveness of its underlying semantics allows dealing with several issues on real-time design.

Most embedded real-time design environments use multiple tools: for 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 in these environments. One solution to this problem consists of using the same semantic model for all design activities.

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 for the design of 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. In this article, 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 involves a homogeneous framework that uses the SIGNAL language. As a result, the formal techniques and tools available within the POLYCHRONY platform can be used for system analysis. Targeting avionics, we consider recent *integrated modular avionics* architectures and their associated standard ARINC [Airlines Electronic Engineering Committee 1997a, 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, that allows us to validate our design choices.

—*Outline.* The rest of the article is organized as follows: Section 2 first presents the SIGNAL language and its associated analysis techniques. Then, Section 3 introduces avionic system architectures. Section 4 describes modeling 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 our study. 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 "no event") may occur between two defined values. It is important to note that here, the symbol $\perp$ does not have the same semantics as in standard denotational semantics where it means "not known yet". In the

semantics of SIGNAL, it means "not known, and never will be known", which is also interpreted as "absence of event" or "undefined value". 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:

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

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 nonpositive 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 nonempty. 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. The special bottom value ($\perp$) is usually employed to represent such a deadlock.

It would be significant if such deadlocks could be statically prevented. For this, it is necessary to be able to statically verify timing properties. Then the $\perp$ should be handled when reasoning about time, but has to be considered with a nonstandard meaning. As mentioned earlier, 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 all synchronization should be completely handled at compile time in such a way that the execution phase has nothing to do with $\perp$. This compilation process is based on 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 tth token (i.e., some value different from $\perp$, carried by a signal) on the first input is evaluated with the tth token on the second input, to produce the tth 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 elaborate constructs for ease of use and structuring. Here, we give a sketch of the primitive constructs (boldfaced) and a few derived constructs (italics) often used. The corresponding syntax and definition are mentioned.

—*Instantaneous functions/relations*: Let `f` be a symbol denoting an n-ary function $[\![f]\!]$ on instantaneous values (e.g., arithmetic or Boolean operation). Then, the SIGNAL expression

```
y:= f(x1,...,xn)
```

defines an elementary process such that

$$\forall t : y_t \neq \perp \Leftrightarrow x1_t \neq \perp \Leftrightarrow \cdots \Leftrightarrow xn_t \neq \perp,\ y_t = f(x1_t, \ldots, xn_t),$$

where xi_k denotes the kth element of the sequence denoted by $\{xi_t\}_{t \geq 0}$. By the notation $e_1 \Leftrightarrow e_2 \Leftrightarrow \cdots \Leftrightarrow e_k$, we mean that the logical expressions $e_1, e_2, \ldots$ and e_k are "all true, or all false".

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

```
y:= x $ 1 init c
```

defines an elementary process such that

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

At the first instant, the signal `y` takes the initialization value `c`. Then, at any instant, `y` takes the previous value of `x`. There is a generalized form of the delay operator `y:= x $ k init array_c` that enables getting the `k`*th* previous value of the signal `x`. Moreover, `k` is a positive integer. The initialization values are provided through an array of constants `array_c`.

—*Undersampling*: 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 when b
```

defines an elementary process such that

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

The derived statement `y:= when b` is equivalent to `y:= b when b`.

—*Deterministic merging*: The unique output provided by this operator is defined (i.e., 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 default y
```

defines an elementary process such that:

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

Note that the preceding functions/relations, undersampling, and merging operators define processes that are characterized without any memorization. The properties of these processes are stated on a single logical time index t. They are referred to as *static processes*. This is not the case for the delay operator, which induces a memorization of previous values of signals. It manipulates two time indices t and $t-1$. The processes defined by this operator are referred to as *dynamic*.

—*Parallel composition*: Resynchronizations (i.e., 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 considering as local signals a subset of the signals defined in a given process. If `x` is a signal defined in a process `P`, the expression

```
P 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 $\texttt{P}_{i,i\in 1..k}$ contain local signals of the same name, these signals are implicitly renamed in the composition of the $\texttt{P}_i$'s so as to distinguish the local signals in each $\texttt{P}_i$.

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

—*Clock extraction*: `h := ^x` specifies the clock `h` of `x`, and can be defined as `h := (x = x)`. Here, `h` is uniformly *true* when `x` is defined, i.e. different from $\perp$. Otherwise, `h` is equal to $\perp$. The equality operator used in the definition is a SIGNAL instantaneous function/relation.

—*Synchronization*: `x1 ^= x2` specifies that `x1` and `x2` have the same clock, and is defined as `h := (^x1 = ^x2) where h`.

—*Clock union*: `h:= x1 ^+ x2` specifies the clock union of `x1` and `x2`, which is also defined as `h:= ^x1 default ^x2`.

—*Memory*: `y:= x cell b init y0` enables memorizing in `y` the latest value carried by `x` when `x` is present or when `b` is *true*. It is defined as

```
y := x default (y$1 init y0) | y ^= x ^+ (when b).
```

2.3 A Simple Example

The following process defines 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*: The form `(|···|)` enables syntactically delimiting the body of a process in SIGNAL. The signal `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`, where `zvdec` is defined as the previous value of `v` (delay), namely `zv`, when this value is present and moreover, when `reset` is present and has the value *false* (operator `when`). Here, the `not` operator is defined by an instantaneous logical negation function/relation. The Boolean signal `reach0` is defined (with the value *true*) when the previous value of `v` is equal to `1`. Note that `v` is decremented when `reset` has a value *false*.

Program abstraction allows us to declare a given process, together with its ways of communication, stated explicitly. As an example, the preceding 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 is 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 a well-defined 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 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 tool 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 when x2
```

expresses the following assertions:

—If `x1` is defined, and `x2` is defined and *true*, then `x3` is defined and `x1 = x3`.
—If `x1` or `x2` is not defined, or `x2` is defined and *false*, then `x3` is not defined.

It appears that useful information (if `x` is a signal) is

—`x` is defined and *false*,
—`x` is defined and *true*, and
—`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 next:

—The equations

$$\mathrm{y}^2 = \mathrm{x}_1^2 = \cdots = \mathrm{x}_n^2$$

denoting the equality of the respective clocks of signals $\mathrm{y}, \mathrm{x}_1, \ldots, \mathrm{x}_n$ are associated with $\mathrm{y} := \mathtt{f}(\mathrm{x}_1, \ldots, \mathrm{x}_n)$ (all static processes are encoded in this way;

however, dynamic processes on Boolean signals are encoded in a slightly differently—see Benveniste and Le Guernic [1990]). Boolean relations may be completely encoded in $\mathcal{F}_3$. For instance, $x_2 = -x_1$ corresponds to $x_2 :=$ `not` x_1: If $x_1 = \mathit{true}$, then $x_1 = 1$ and $-(x_1) = -1$, which is associated with *false*.

—The equation

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

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

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

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

—The equation

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

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, that is, 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, that is, 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

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

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 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

$$c^2 = a^2$$
$$x^2 = a^2(-c - c^2)$$
$$y^2 = x^2 = a^2,$$

which results in $c^2 = a^2 = y^2 = x^2 = a^2(-c - c^2)$ and by substitution

$$c^2 = c^2(-c - c^2) \text{ and then } c = 1 \text{ or } c = 0.$$

This means that 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

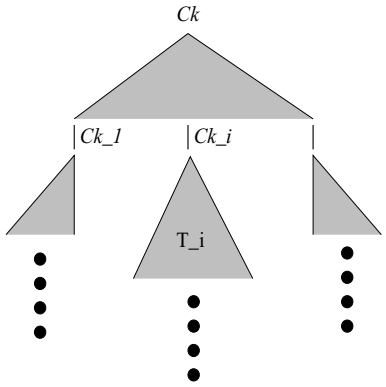


Fig. 1. Clock hierarchy of an endochronous program.

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, 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 undersampling 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, namely, an undersampling 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 undersamplings 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 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]. Figure 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, we can derive several "endochronous" subtrees (e.g., 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 undersamplings (otherwise, the clock is a root).

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

2.4.3 *An Example.* Consider again the process RCOUNT of Section 2.3. The clock calculus finds the clocks

$$\begin{gathered}\mathtt{reset}^2\\ \mathtt{vreset}^2 = -\mathtt{reset} - \mathtt{reset}^2\\ \mathtt{v}^2 = \mathtt{zv}^2 = \alpha^2 = (-\mathtt{reset} - \mathtt{reset}^2) + (\mathtt{reset} - \mathtt{reset}^2)\mathtt{v}^2\\ \mathtt{vdec}^2 = \mathtt{zvdec}^2 = \mathtt{v}^2(\mathtt{reset} - \mathtt{reset}^2)\\ \mathtt{reach0}^2 = -\alpha - \mathtt{v}^2,\end{gathered}$$

where α is the coding of $\mathtt{zv} = 1$.

The clock calculus does not synthesize a master clock for this process. In fact, it is not endochronous (and is nondeterministic): When reset is *false*, then zvdec is defined if zv is defined, namely, if v is defined; but v is defined (when reset is *false)* if vdec is defined, In other words, if zvdec is defined 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 successive composition operations from inner to outer levels. 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))
 | (reach0, v) := RCOUNT {v0}(reset)
 |)
```

```
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 has the value *true* exactly when reach0 is present. There is a master clock ($h^2 = v^2 = reset^2$) 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 the 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 those 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 next.

The *graph of conditional dependencies* (GCD) calculated by the SIGNAL compiler for a given program is a labeled directed graph where:

—vertices are the signals plus clock variables,
—arcs represent dependence relations, and
—labels are polynomials on $\mathcal{F}_3$ which represent 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.

$y := f(x_1, \ldots, x_n)$	$y^2 : x_1 \rightarrow y, \ldots, \ y^2 : x_n \rightarrow y$
$y :=$ x when b	$y^2 : x \rightarrow y, \ y^2 : b \rightarrow y^2$
$z :=$ x default y	$x^2 : x \rightarrow z, \ y^2 - x^2y^2 : y \rightarrow z$

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 never occur during execution. 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 of the aforementioned properties checked by the SIGNAL compiler during the clock calculus are mainly static. For more information on issues related

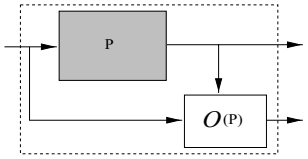


Fig. 2. Composition of a program P together with its observer $\mathcal{O}$(P).

to the clock calculus, the reader may refer to Amagbegnon et al. [1994, 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 this, 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" 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 enumerating 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 checking 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 the 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 the considered properties have been satisfied (this can be expressed, e.g., through Boolean output signals such as 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, like 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.

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

The POLYCHRONY environment [ESPRESSO-IRISA 2006] associated with the SIGNAL language offers several functionalities, including all the facilities mentioned in previous 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. Figure 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.

—*Integrated modular avionics (IMA)*: Recent IMA architectures propose a new way to deal with the 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 Figure 4). They are guaranteed a safe allocation of shared resources so that no fault propagation occurs from one component to another. This is achieved through *partitioning* the resources with respect to available CPU, memory, and bandwidth.

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 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 transmitted via the channel or overwritten by a new occurrence of the message. A received message remains in the destination

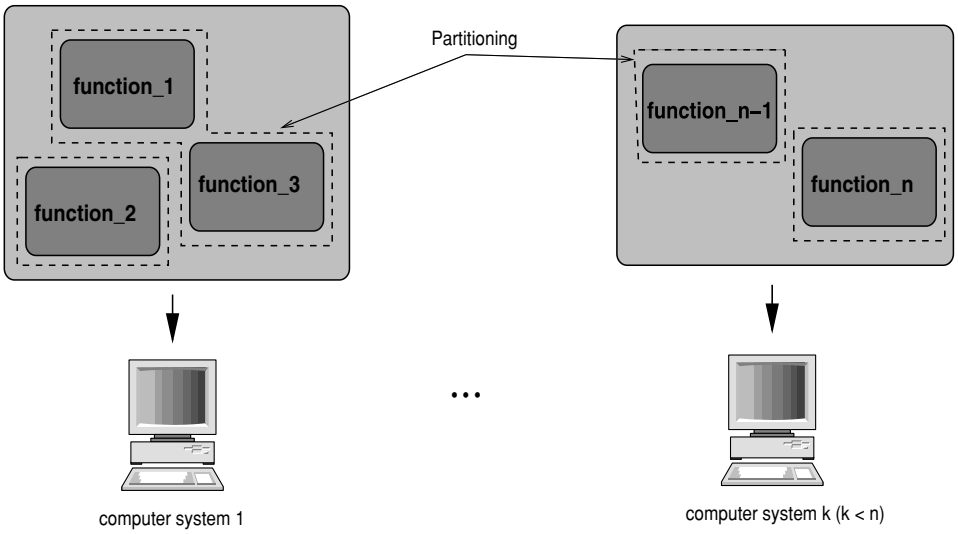


Fig. 4. Integrated modular avionics. Different functions can share a fault-tolerant computer.

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, 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).

Partitions are composed of processes that represent the executive units.[2] Processes run concurrently and execute functions associated with the partition in which they are contained. Each process is uniquely characterized by information (e.g., 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* enables sending and receiving 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 using a *semaphore*. Figure 5 illustrates an example of IMA partitioning.

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 and is called the *avionics application software standard interface* (see Figure 6). It includes services for communication between partitions on the one hand, and between processes on the other, as well as synchronization

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

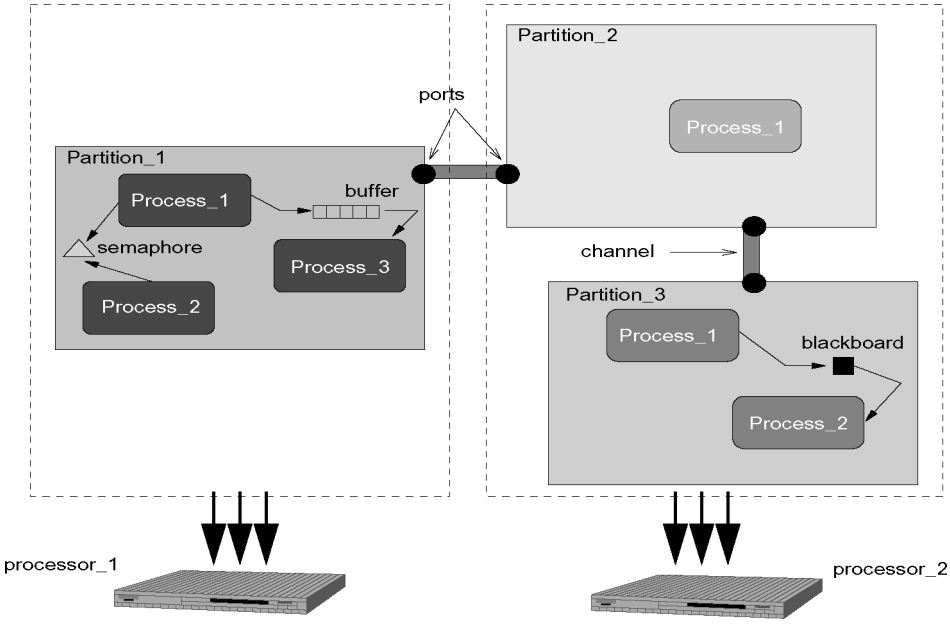


Fig. 5. An example of application partitioning.

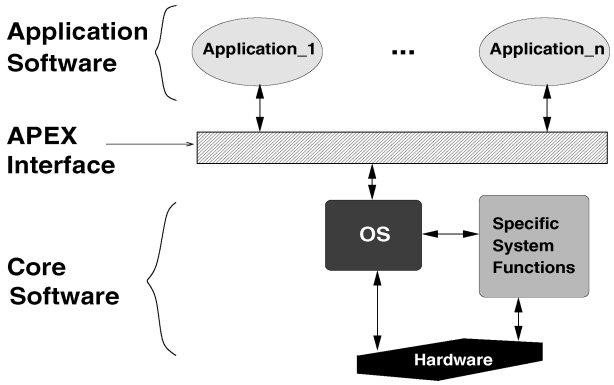


Fig. 6. The APEX interface within the core module software.

services for processes, partition and process management services, and time and error management services.

4. MODELING 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);

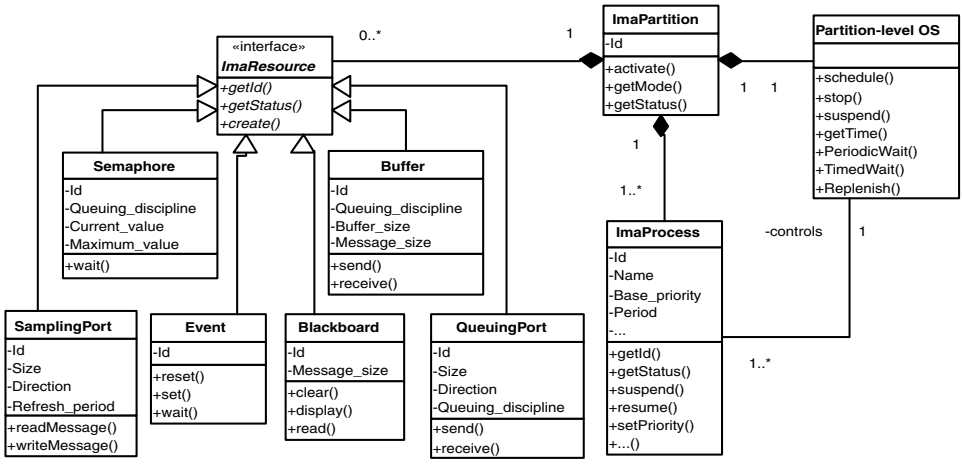


Fig. 7. Basic building blocks for partition modeling.

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

These building blocks are summarized in the UML diagram illustrated in Figure 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 Figure 8.

In the following, modeling 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*: Figure 9 shows an abstract formal specification corresponding to the service. This specification mainly expresses 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 (i.e., `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

```
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.

```
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.

empty and the value of the input parameter `timeout` 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* 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 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 --> 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 Figure 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. This leads to ambiguities which are not 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 that 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 *resumes* (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 resumed.

The level of detail of the model described in Figure 9, although very abstract, enables covering 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 less expensive 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 this 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 this case, only the latest occurrences of messages are relevant.

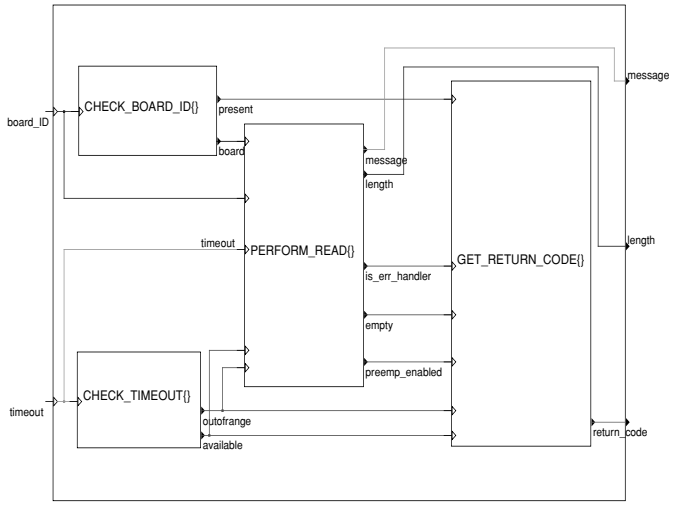
The presence of ambiguities as we have just illustrated justifies a model refinement design approach: Abstract descriptions are progressively refined in order to derive particular implementations. Here, the way messages are retrieved during the *read_blackboard* service call is not fixed (there are two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

In the next step, the more general model defined previously is refined by precising 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 illustrated in Figure 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 Figure 10 gives more details 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 subparts are distinguished based on the service informal specification. They are represented by inner boxes in the graphical description (see Figure 10). The subparts `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 subparts of the service model. Each subpart 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 subparts. For instance, this is what Eqs. `(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. In fact, we have now enough information to be able to determine its values. This is




```
(| 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.

described by Eq. (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 subpart. Finally, a complete SIGNAL specification of the service is obtained, as illustrated in Gamatié and Gautier [2002]. 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 purposes, 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 presents the modeling of the partition-level OS which is in charge of controlling the execution of processes within a partition.

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

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 Figure 11.

The notions taken into account for modeling 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 earlier. 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 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 (compare with Figure 11). We explain how it interacts with processes within a partition.

In Figure 11, the input `Active_partition_ID` represents the identifier of the running partition selected by the module-level OS, and 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 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 `timedout` 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 `timedout`.

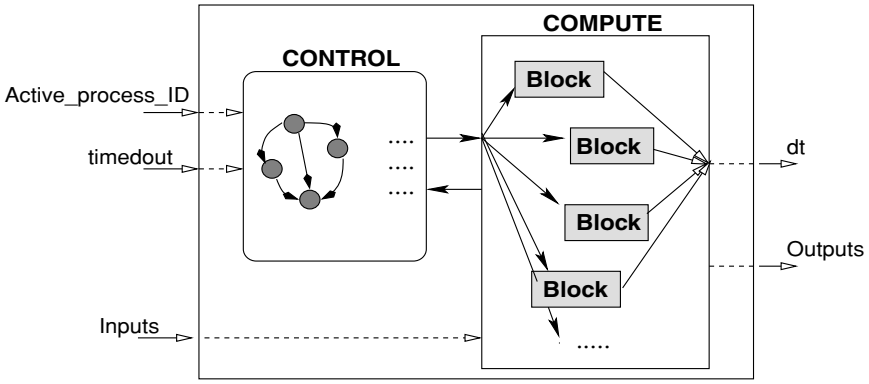


Fig. 12. ARINC process model.

4.3 IMA Processes

The definition of an IMA process model basically takes into account its computation and control parts. This is depicted in Figure 12. Two subcomponents 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 `timedout` notifies processes of time-out expiration. In addition, there are other inputs (respectively, outputs) needed for (respectively, produced by) the process computations. CONTROL and COMPUTE subcomponents cooperate to achieve the correct execution of the process model.

The CONTROL subcomponent 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 subcomponent is selected to be executed *instantaneously* (this is represented by the arrow from CONTROL to COMPUTE in the figure).

The COMPUTE subcomponent 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, we 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.

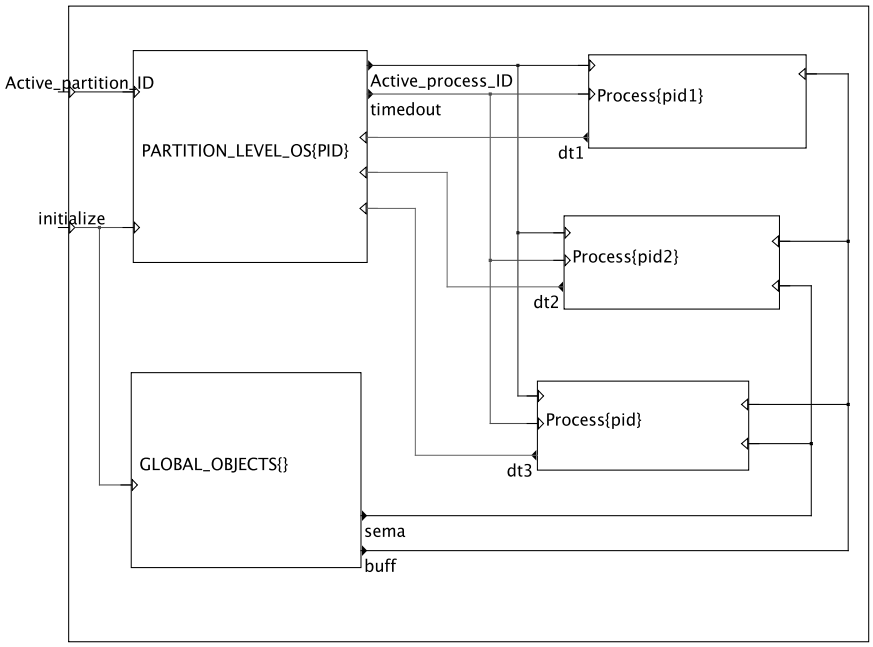


Fig. 13. An example of a partition model composed of three processes.

This restriction could be avoided by using the appropriate control mechanisms of SIGNAL. Typically, a Boolean state variable could be introduced within each block. Then, the execution of statements would be undersampled by values of this variable. This solution potentially increases the number of state variables in a program and is therefore expensive. We can imagine further possibilities, but for the sake of simplicity, we rather advocate to avoid the combination of system calls and other statements in the same block.

4.4 Partitions

Figure 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, the communication and synchronization mechanisms used by processes (e.g., `buff`, `sema`) are created there.

The UML sequence diagram[3] depicted in Figure 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 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, the following action is performed: The process executes a block from its COMPUTE subpart, and the duration corresponding to the executed block is

[3]UML Specification version 2.0: Superstructure—Object Management Group (`www.omg.org`).

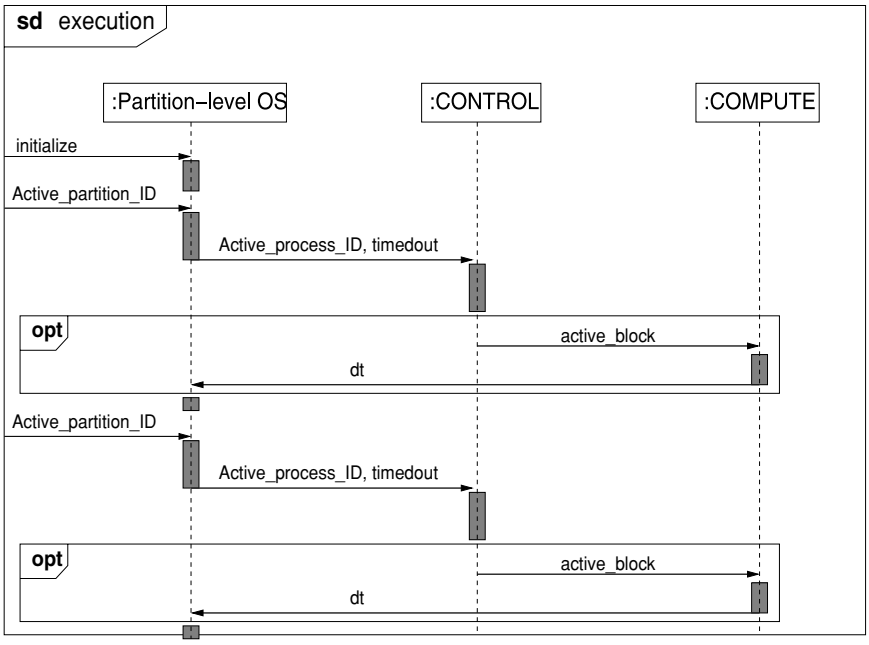


Fig. 14. A sketch of the model execution.

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.

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 nonfunctional 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 next were 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

$$P = P_1 \mid P_2 \mid \cdots \mid P_n,$$

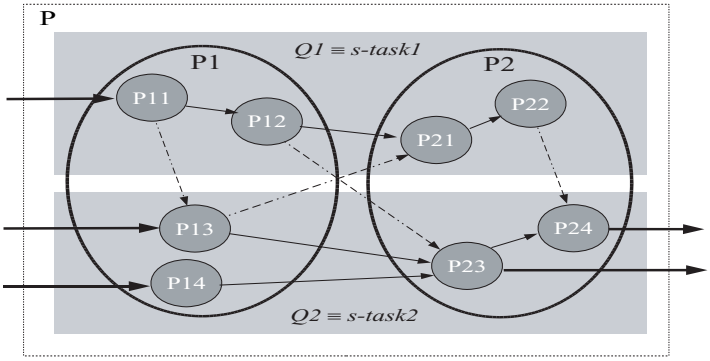


Fig. 15. Decomposition of a SIGNAL process into two s-tasks Q_1 and Q_2.

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

(1) Considered programs P are endochronous (see Section 2.4.2), hence deterministic;
(2) they only contain definitions with acyclic dependencies,
(3) a set of processors $q = \{q_1, q_2, \ldots, q_m\}$, and
(4) a function $locate : \{P_i\} \longrightarrow \mathcal{P}(q)$, which associates with each subpart of an application $P = P_1 \mid P_2 \mid \cdots \mid P_n$ a nonempty 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 Figure 15. Each subprogram P_i (represented by a circle) is itself composed of four subprograms P_{i1}, P_{i2}, P_{i3}, and P_{i4}. The program P is distributed on two processors q_1 and q_2 as follows:

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

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 $Q_2 = P_{13} \mid P_{14} \mid P_{23} \mid P_{24}$:

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

Remark 5.1. The aforementioned transformation remains valid even if $locate(P_{ik})$ is not a singleton. In that case, P_{ik} is split into new subprograms 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

$$\begin{aligned} P &= P_{11} \mid P_{13} \mid P_{12} \mid P_{14} \mid P_{21} \mid P_{22} \mid P_{23} \mid P_{24} \ (P_{11\_13} \text{ is split}) \\ &= P_{11} \mid P_{12} \mid P_{13} \mid P_{14} \mid P_{21} \mid P_{22} \mid P_{23} \mid P_{24} \ (\text{commutativity of } \mid) . \\ &= P_1 \mid P_2 \end{aligned}$$

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

The subprograms 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 multiprocessor 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 previous transformation is how hidden signals are handled. The obvious concern is name capture. For instance, let us consider the example from Figure 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 composition of such processes. More generally, for two different processes, unless there is an explicit communication (through interface signals) between them, their local signals having the same name designate different signals. Therefore, these local signals are systematically renamed differently during composition.

Second transformation. We want to refine the level of granularity resulting from the preceding transformation. For that, let us consider descriptions at the processor level (i.e., s-tasks). We are now interested in how to decompose s-tasks into fine-grained 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 that the nodes' execution completes entirely 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 (respectively, 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 (respectively, on the occurrence of i). In that case, all possible intermediate nodes are also executed.

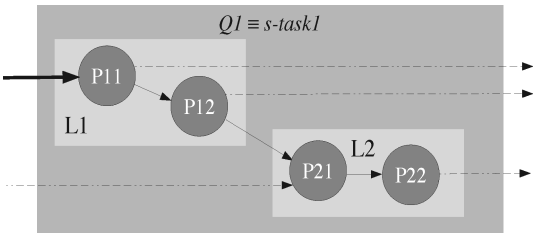


Fig. 16. Decomposition of an s-task into two clusters L_1 and L_2.

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.

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 subtree of the global clock tree characterizing the program. Hence, this ensures a deterministic execution of each cluster. Figure 16 shows a decomposition of the s-task Q_1 into two clusters L_1 and L_2. The input of the subprogram $P11$ (boldfaced 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 and resulting programs is strictly preserved.

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

5.2 Instantiation of SIGNAL Programs in the IMA Model

We first present this instantiation at processor level then the approach can be generalized to the multiprocessor level. From the aforementioned 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* 1: *Distribute the program on the available processors.* Here, we assume a given distribution function. The program is transformed into s-tasks. 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* 2: *For each processor, transform the associated s-task into a graph of clusters.* This task is automatically performed by the SIGNAL compiler.

—*Step* 3: *For each processor, associate clusters with partitions/processes.* The first decision about the graph of clusters resulting from the previous step

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

Fig. 17. Modeling rules of IMA partitions.

consists of 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 $Q1$ (compare to Figure 16) by one partition. Once partitions are chosen, the graph corresponding to each of them is decomposed into subgraphs. These contain the clusters that should be executed by the same process. In the example, clusters associated with "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., subsets of clusters) which strongly depend on each other are also put together in the same partition. This repartition has the advantage of greatly reducing the interprocess communication costs. In the next step, the program can be effectively instantiated using our library of models (see Section 4).

—*Step* 4: *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 Figure 17. The symbol "|" denotes the synchronous composition. The following transformations are defined:

(1) Description of the process associated with a set of clusters:
   (a) The definition of the CONTROL part of the process relies on dependencies between clusters. Clusters are executed sequentially with respect to these dependencies.
   (b) Each cluster is "embedded" in a block within the COMPUTE part of the process.
   (c) The internal communications between the clusters of a subgraph associated with a process are modeled using local state variables (i.e., those defined by the delay construct). These variables enable memorizing exchanged data. On the other hand, communications between subgraphs of clusters from different processes are modeled with APEX services. For each entry point (respectively, exit point) of a subgraph, 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 (respectively, after) the block that contains the cluster concerned by the entry point (respectively, the exit point). The choice of suitable service to call here depends on the desired type of communication. The program by itself is not enough to decide which

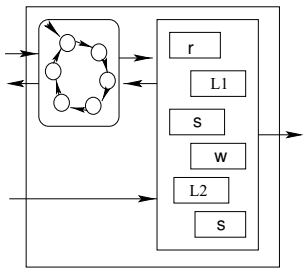


Fig. 18. IMA model associated with $Q1$ after transformation.

service is better-suited. 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 preferable to those related to a blackboard. By contrast, if the communication only requires one memory-place, the blackboard will be more appropriate. Events are preferable to the two previous communication mechanisms when the exchanged data is specifically used to notify a process about a certain condition. Services associated with semaphores are used for synchronization.

(2) Description of the partition associated with a set of clusters:
   (a) 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.
   (b) 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` subpart of the partition, as illustrated in Figure 13.

Example. In Figure 18, we outline a process model resulting from the transformation of $Q1$. There are six blocks where two contain 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 in which blocks are added is determined by the structure of $Q1$. But the choice of communication mechanism (e.g., buffer or blackboard for the block `r`) contained in these blocks requires expert decisions. For instance, the process is here supposed to wait for some event notification between `L1` and `L2`, thus we use `w` instead of `r`, etc. The automaton described in the control part gives the execution order of blocks with respect to the precedence constraints of the cluster graph. The corresponding partition is obtained by considering the phase 2 of Step (4), in the current section of the article.

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, but the scheduling policy differs.

Our approach for partitioning is very interesting because it uses the formal transformations [Sacres Consortium 1997] defined in SIGNAL in order to be able to guarantee the functional correctness-by-construction of resulting applications. However, the criticality issue in these applications remains difficult. It generally requires domain expert knowledge that could be, of course, integrated in our approach for further improvements. In current industrial practices, avionic functions with high critical level are still designed using federated architectures (e.g., Airbus A380). The IMA part mostly concerns less critical functions where the criticality issue is not very crucial. A reason to that is the lack of clear guidelines that help designers to cope with the criticality issue. Even if our approach, as many others (see Section 7), aims to propose a solution to this problem, we believe that consortia including both the avionics industry and academia offer more suitable frameworks to define complete solutions.

To conclude this section, we briefly mention another study in which our ARINC concept models have been used [Talpin et al. 2003]. We defined a technique in order to import resource-constrained, multithreaded, 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 enables 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 the remapping of threads onto a given distributed architecture or prescribed target real-time operating system. This permits 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 enables checking for the absence of cyclic definitions in a program or in the endochrony property of this program. These properties are necessary if we need to generate the code associated with the program, for example, 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, however, we concentrate on timing issues. We expose a high-level technique, entirely defined in SIGNAL, that allows us 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 of using an observer program, referred to as program *interpretation* in the sequel, to check

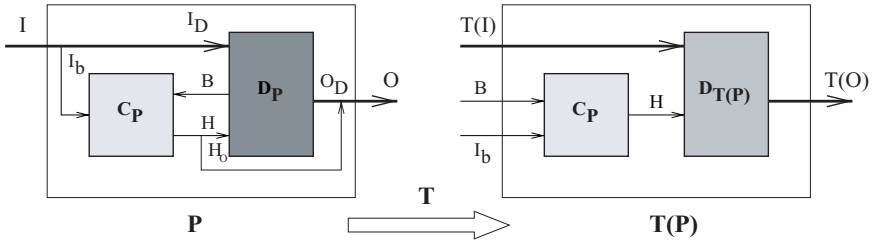


Fig. 19. Temporal interpretation of a SIGNAL process P.

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 checking 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 us 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 effectively explored before committing the design to one particular implementation. Such an approach permits concentrating 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

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

The process of obtaining an interpretation $T(P)$ of a process P is graphically depicted in Figure 19. 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

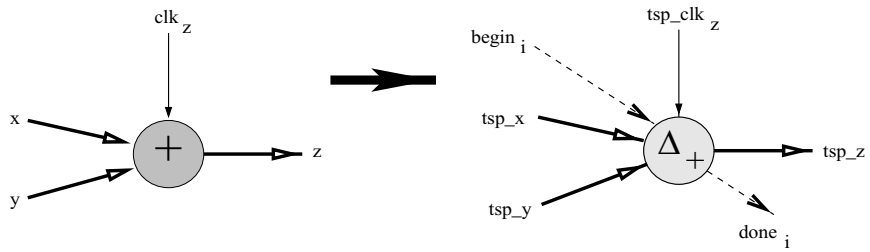


Fig. 20. Node associated with `z := x + y` (left); and its temporal model (right).

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 subprocesses, where elementary subprocesses 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 counterparts. The resulting time model is composed with the original functional description of the application (using 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 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 sufficient to consider more abstract representations, either of the program or of its temporal model.

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 20, on the lefthand-side. Besides the input values `x` and `y`, this node also requires clock information, denoted by the signal `clk_z`, which triggers computation of the output value `z`. The associated temporal model is represented by the node illustrated on the righthand-side of the figure. The SIGNAL program corresponding to this temporal model, called `T_Plus`, is depicted by Figure 21. 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 can be executed. The presence of `begin_i` in a node means that the preceding nodes (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.,

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

`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 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.

In addition to the library of cost functions of primitive constructs, the implementation also requires *platform-dependent information* (e.g., the delay of 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 shown in Figure 21, this information is 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 shown in Gamatié et al. [2004]. Figure 22 shows a *cosimulation* of this application together with its temporal interpretation T_SATMAINT (the prefix notation "T_" stands for temporal).

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 $[c_1, \ldots, c_q]$ computed in the original program (represented by B in Figure 19). The Boolean c_i are mainly signals 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 yields timestamp information. The vector $[c_1, \ldots, c_q]$ contains intermediate Boolean signals evaluated by the data part of SATMAINT, which are needed by T_SATMAINT to compute output information. Note that in a straightforward approach, it is possible to provide a set of

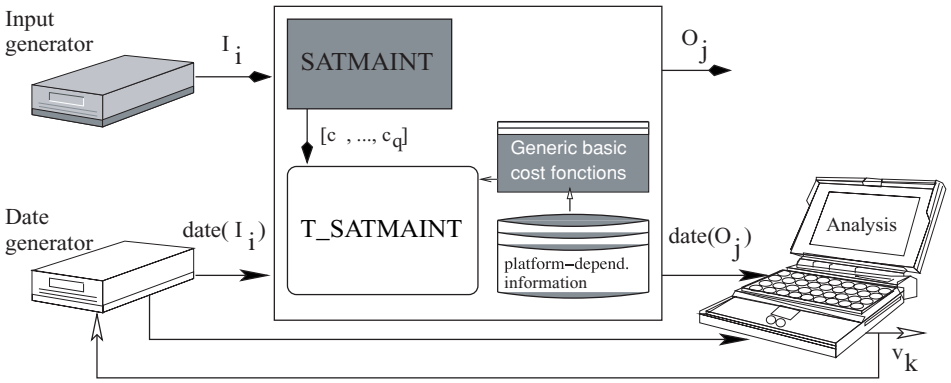


Fig. 22. Cosimulation of SATMAINT with its temporal interpretation.

vectors that covers all 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 22. This 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, 1995] where, the 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 of using a modular evaluation approach which is exposed in the sequel.

The modularity of the SIGNAL language allows constructing 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 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 $\mathtt{P} \equiv \mathtt{P}_1 \mid \cdots \mid \mathtt{P}_n$ denote the program corresponding to the application considered for temporal analysis. We want to define the program $\mathtt{P'} \equiv$ `(| P | T_P|)` that is used for the analysis, where `T_P` is the temporal model of `P`. The same program can be also rewritten as: $\mathtt{P'} \equiv \mathtt{P'}_1 \mid \cdots \mid \mathtt{P'}_n$. Each $\mathtt{P'}_i$ denotes the

composition of a subprogram $\mathtt{P}_i$ of P, with its associated temporal interpretation $\mathtt{T\_P}_i$. The following steps are identified in order to carry out experiments:

(1) *Partial definition of temporal interpretations*: For each subprogram $\mathtt{P}_{i,i\in\{1,\ldots,n\}}$ of P, we define the corresponding temporal model $\mathtt{T\_P}_i$.
(2) *Composition of each subpart of the application with its associated temporal interpretation, then abstraction*: This step defines the subprograms $\mathtt{P'}_i$ ($i \in \{1,\ldots,n\}$) which constitute the simulation program P' that we want to construct. The abstraction aims to keep only relevant information of these subprograms for the cosimulation. It follows that $\mathtt{P'}_i \equiv \alpha(\mathtt{P}_i \,|\, \mathtt{T\_P}_i)$, where α denotes the abstraction (e.g., a program can be abstracted by considering only its control part, namely Boolean and synchronization signals, or by approximating the value of numerical signals, for instance, by dealing with domains of intervals instead of pointwise domains).
(3) *Construction of the global model for simulation*: This model results from the composition of subprograms $\mathtt{P'}_i$ defined at the previous step, that is,

$$\mathtt{P'} \equiv \mathtt{P'}_1 \mid \cdots \mid \mathtt{P'}_n.$$

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

$$\mathtt{T\_COMPUTE} \equiv \mathtt{T\_BLOCK\_0} \mid \cdots \mid \mathtt{T\_BLOCK\_7}.$$

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

$$\mathtt{T\_PROC\_8} \equiv \mathtt{T\_COMPUTE\_8} \mid \mathtt{T\_CONTROL\_8}.$$

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

$$(\mid \mathtt{PROC\_8} \mid \mathtt{T\_PROC\_8} \mid).$$

This process can now be simplified by considering approximations (or abstractions). This facilitates 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 is clock relations and dependencies between the signals of P. Hence, the resulting program could be considered for cosimulation 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 constant δ to the timestamps corresponding to input availability in order to compute timestamps associated with outputs. This interpretation is simpler than that obtained by composing the temporal models of all intermediate operations carried out by the considered subprogram. We can also mention other simplifications, which consist of considering worst-case execution times (WCET) [Puschner and Burns 2000] in the definition of the temporal interpretation of a subprogram. Over 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, we may use a range of possible execution times. This last possibility provides more accurate information, since the execution order 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 first specifying, either completely or partially (by using abstractions), subparts 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 this, we used a technique consisting of cosimulating 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 Applications 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 characteristic common to both GIOTTO and synchronous formalisms is that they both enable platform-independent specifications.

Component-based design techniques have revealed advantages, reusability 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 first, 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 components to active runtime structures (such as tasks). A set of analysis tools is also available for real-time and reliability analysis. To enable this, a library of microcomponents, components, and infrastructures has been defined. Microcomponents 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 a 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 used for the design of distributed embedded real-time systems. The AAA methodology covers the entirety of design steps, from the specification of functionalities (i.e., the software level) to deployment on target multiprocessor 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 criteria are to satisfy timing requirements and minimize hardware resources). The distribution relies on graph representations, and takes nonfunctional requirement inputs (e.g., real-time performances). Finally, the 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 system: the structure of a system (its components and communication between them), 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 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 an agent (a building block) and mode (a 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 interaction 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 of providing the designer with a methodology, an architecture description language (ADL) called *Cotre*, and an environment in which 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), developed by the International Society of Automotive Engineers (SAE) [AADL Coordination Commitee 2002]. It is dedicated to the design of software and hardware components of an avionic system and the interfaces between these components. The AADL definition is based on METAH (an ADL developed by Honeywell) [Vestal 1997]. It permits 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 the avionics domain [Camus and Dion 2003]. It is also based on 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 the drastic savings in 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] the authors 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 for a large amount of jitter and discuss possible ways to reduce this by checking process periods with respect to partition 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 (e.g., the future Airbus A380). This is likely due to the fact that partitioning raises several questions that are not yet sufficiently addressed. Among these questions, we can mention the correctness of a global partitioning, which is crucial. A formal description of partitioning requirements is proposed by 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 defining a correct-by-construction partitioning based on 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 [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 [2003].

8. CONCLUSIONS

In this article, 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 runtime 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 reusability 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 in this regard arises from the partitioning itself in such systems. We believe that the ideas exposed in this article help to overcome this difficulty. For instance, the sensitivity analysis we use in our approach allows us to easily identify dependencies between subparts 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* 1 (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 article 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]. These turn out to promote *globally asynchronous locally synchronous* architectures (GALS) where a system is composed of subsystems/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. These are freely available together within the POLYCHRONY platform at ESPRESSO-IRISA [2006]. 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 and anonymous reviewers for their valuable comments on the previous versions of this article.

REFERENCES

AADL Coordination Commitee. 2002. Avionics architecture description language. In *AADL Seminar* (Toulouse, France). Society of Automotive Engineers.

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. `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 Language Design and Implementation (PLDI)*. ACM Press, New York. 163–173.

Arvind and Gostelow, K. 1978. *Some Relationships Between Asynchronous Interpreters of a Dataflow Language*. North-Holland, New York.

Audsley, N. and Wellings, A. 1996. Analysing APEX applications. In *Real Time Systems Symposium (RTSS)* (Washington, DC). IEEE Press.

Benveniste, A. 1998. Safety critical embedded systems design: The SACRES approach. In *School on Formal Techniques in Real Time and Fault Tolerant Systems 1998: Material for the School*. Technical University of Denmark, Lyngby, Denmark.

Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and de Simone, R. 2003. The synchronous languages twelve years later. *Proc. IEEE 91*, 1 (Jan.), 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 *Proceedings of the 2nd International Conference on Embedded Software (EMSOFT)* (London). Springer Verlag. 252–265.

Benveniste, A. and Le Guernic, P. 1990. Hybrid dynamical systems theory and the signal language. *IEEE Trans. Autom. Control 35*, 5 (May), 535–546.

Berry, G. and Sentovich, E. 1998. Embedding synchronous circuits in GALS-based systems. In *Sophia-Antipolis Conference on Micro-Electronics (SAME)*. Sophia Antipolis MicroElectronics, 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. *Electron. Not. Theor. Comput. Sci. 80*, 1–16.

Bodin, F. and Puaut, I. 2005. A WCET-oriented static branch prediction scheme for real time systems. In *Proceedings of the 17th Euromicro Conference on Real Time Systems (ECRTS)*. 33–40.

Camus, J.-L. and Dion, B. 2003. Efficient development of airborne software with Scade suite. Tech. Rep., Esterel Technologies. `www.esterel-technologies.com`.

Caspi, P. 2001. Embedded control: From asynchrony to synchrony and back. In *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT)* (Lake Tahoe, CA), Th. A. Henzinger and Ch. M. Kirsch, Eds. Lecture Notes in Computer Science, vol. 2211, Springer Verlag.

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 the Computer Aided Verification (CAV)* (London). Springer Verlag.

Conmy, P. and McDermid, J. 2001. High level failure analysis for integrated modular avionics. In *Proceedings of the 6th Australian Workshop on Safety Critical Systems and Software (SCS)*. Australian Computer Society, Darlinghurst Australia. 13–21.

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)* (Pisa, Italy). IEEE Press.

DENNIS, J. B., FOSSEN, J. B., AND LINDERMAN, J. P. 1974. Data flow schemas. In *Proceedings of the International Symposium on Theoretical Programming* (London, UK), A. Ershov and V. A. Nepomniaschy, Eds. Lecture Notes in Computer Science, vol. 5, Springer Verlag. 187–216.

DI VITO, B. 1999. A model of cooperative noninterference for integrated modular avionics. In *Proceedings of the Conference on Dependable Computing for Critical Applications (DCCA)* (Washington, DC). IEEE Computer Society. 269.

ESPRESSO-IRISA. 2006. The POLYCHRONY website. `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. `www.inria.fr/rrrt/rr-4678.html`.

GAMATIÉ, A. AND GAUTIER, T. 2003a. The signal approach to the design of system architectures. In *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*. 80.

GAMATIÉ, A. AND GAUTIER, T. 2003b. Synchronous modeling of avionics applications using the SIGNAL language. In *9th IEEE Real-Time/Embedded Technology and Applications Symposium*. IEEE Computer Society, Los Alamitos, CA.

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)* (Gothenburg, Sweden). Lecture Notes in Computer Science. Springer Verlag.

GAUTIER, T. AND LE GUERNIC, P. 1999. Code generation in the SACRES project. In *Safety-Critical Systems Symposium (SSS)* (Huntington, UK), F. Redmill and T. Anderson, Ed. Springer Verlag.

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 (MEMOCODE)*. IEEE Computer Society, Los Alamitos, CA.

HALBWACHS, N. AND BAGHDADI, S. 2002. Synchronous modelling of asynchronous systems. In *Proceedings of the Conference on Embedded Software (EMSOFT)* (Grenoble, France), J. Sifakis and A. Sangiovanni-Vincentelli, Eds. Lecture Notes in Computer Science, vol. 2491, Springer Verlag. 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 25th International Conference on Software Engineering (ICSE)* (Washington, DC). IEEE Computer Society. 160–173.

HENZINGER, T., HOROWITZ, B., AND KIRSCH, C. 2001. Embedded control systems development with Giotto. In *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems*. ACM Press, New York. 64–72.

KAHN, G. 1974. The semantics of a simple language for parallel programming. In *Information Processing*, vol. 74, J. L. Rosenfeld, Ed. North-Holland, New York. 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 *IEE Colloquium on Hardware-Software Cosynthesis for Reconfigurable Systems*. (Jun. 1–9), Bristol, UK.

LE GUERNIC, P., TALPIN, J.-P., AND LE LANN, J.-C. 2003. Polychrony for system design. *J. Circ. Syst. Comput. 12*, 3 (Apr.), 261–304.

LEE, E. 2000. What's ahead for embedded software? *IEEE Comput. Mag. 33*, 9 (Sept.), 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 (DSN)* (Washington, DC). IEEE Computer Society. 14–23.

MARCHAND, H., BOURNAI, P., LE BORGNE, M., AND LE GUERNIC, P. 2000. Synthesis of discrete-event controllers based on the SIGNAL environment. *Discrete Event Dynam. Syst. Theory Appl. 10*, 4 (Oct.), 325–346.

PNUELI, A. 2002. Embedded systems: Challenges in specification and verification. In *Proceedings of the 2nd International Conference on Embedded Software (EMSOFT)* (Grenoble, France), J. Sifakis and A. Sangiovanni-Vincentelli, Eds. Lecture Notes in Computer Science, vol. 2491. Springer Verlag. 252–265.

PUAUT, I. AND DECOTIGNY, D. 2002. Low-Complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)* (Washington, DC). IEEE Computer Society. 114.

PUSCHNER, P. AND BURNS, A. 2000. A review of worst-case execution-time analysis. *J. Real-Time Syst. 18*, 2-3 (May), 115–128.

ROMBERG, J. 2002. Model-Based deployment with autofocus: A first cut. In *Proceedings of the 14th Euromicro Conference on Real Time Systems (ECRTS)* (Work in Progress session). IEEE Computer Society, Los Alamitos, CA. 41–44.

RUSHBY, J. 1999. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Tech. Rep. CR-1999-209347, NASA Langley Research Center. June. `www.csl.sri.com/users/rushby/partitioning.html`.

SACRES CONSORTIUM. 1997. The semantic foundations of SACRES. Tech. Rep. EP 20897, Esprit Project EP. March.

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 (ISICT)* (Dublin, Ireland). 579–584.

SHARP, D. AND ROLL, W. 2003. Model-Based integration of reusable component-based avionics systems. In *Workshop on Model-Driven Embedded Systems in RTAS*. IEEE Computer Society, Los Alamitos, CA.

SIFAKIS, J. 2001. Modeling real-time systems—Challenges and work directions. In *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT)* (London). Springer Verlag.

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 and Embedded Technology and Applications Symposium (RTAS)* (Washington, DC). IEEE Computer Society. 58.

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 *Proceedings of the International Workshop on Scientific Engineering of Distributed JAVA Applications* (Berlin). Springer Verlag. 33–47.

VESTAL, S. 1997. MetaH support for real-time multi-processor avionics. In *Workshop on Parallel and Distributed Real-Time Systems*. IEEE Computer Society, Los Alamitos, CA.

WADGE, W. W. 1979. An extensional treatment of dataflow deadlock. In *Semantics of Concurrent Computation*, G. Kahn, Ed. Lecture Notes in Computer Science, vol. 70, Springer Verlag. 285–299.

WIRTH, N. 2001. Embedded systems and real-time programming. In *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT)* (Tahoe City, CA), Th. A. Henzinger and Ch. M. Kirsch, Eds. Lecture Notes in Computer Science, vol. 2211. Springer Verlag. 486–492.

Received March 2005; revised August 2006; accepted October 2006