Modeling of Avionics Applications and Performance Evaluation Techniques using the Synchronous Language SIGNAL*

Abdoulaye GAMATIÉ, Thierry GAUTIER, Loïc BESNARD
{*agamatie, gautier, lbesnard*}@*irisa.fr*

IRISA / INRIA / CNRS
F-35042 RENNES, France

Abstract Modeling is widely accepted to be essential to all design activity. A major benefit is that formal methods can be applied for analysis and predictability. In POLYCHRONY, the SIGNAL language tool-set, we have defined a component-based approach to the modeling of avionics applications. A library of so-called APEX services relying on the avionics standard ARINC 653 has been defined in the SIGNAL model. This allows the access to the formal tools and techniques available within POLYCHRONY for verification and analysis.
In this paper, we illustrate this approach by considering a small example avionics application. We show how an associated SIGNAL model is obtained for the purpose of temporal validation. This brings out the capability of the SIGNAL language to seamlessly address critical issues in the design of real-time systems.

Keywords *Component-based Modeling, Avionics applications,* SIGNAL*, Temporal Validation.*

1 Introduction

Today, in the design of embedded systems such as avionics systems, key challenges are typically the correctness of the design with respect to the requirements, the development effort and time to market, and the correctness and reliability of the implementation. So, one can observe the need of a seamless design process which takes into account these challenges. In such a context, modeling plays a central role. Among its advantages [15], we mention the enhanced adaptability of models and their parameters; more general descriptions by using genericity, abstraction, behavioral non determinism, and the possibility of applying formal methods for analysis and predictability.

Several model-based approaches have been proposed [16] [11] [13] [2] for the development and verification of embedded systems. They use different kinds of formalisms for the modeling and provide tools for system development and validation. While our approach aims at the same objective, its main particularity relies on the use of a single semantical model, SIGNAL [9], to describe embedded applications from

* This work has been supported by the european project IST SAFEAIR (Advanced Design Tools for Aircraft Systems and Airborne Software) [10].

specification to implementation with the possibility of verification and analysis. This facilitates the validation. POLYCHRONY, the tool-set for SIGNAL, which is developed by INRIA[1] (`http://www.irisa.fr/espresso/Polychrony`), offers the required functionalities (high level specifications, modular verification and analysis, automatic code generation, etc.).

The study presented in this paper is part of a more general design methodology for distributed embedded applications, defined earlier during the SACRES project [8] and currently improved. This methodology is based on an iterative application of transformations on a SIGNAL model that preserve semantic properties. During the transformations, "abstract" components can be instantiated in different ways from modules related to actual target architecture features, addressing various purposes (e.g. embedded code generation, temporal validation). In this context, a library of specific components has been defined in SIGNAL. It includes on the one hand elementary communication mechanisms like FIFOs [7], and on the other hand more complex models like those presented in [6] for the description of avionics applications based on the ARINC standard. In particular, we illustrate here how the SIGNAL model corresponding to an avionics application can be specified using these components in order to perform analysis. As such analysis, we present the techniques implemented in POLYCHRONY for performance evaluation.

The remainder of the paper is organized as follows: Section 2 first discusses the ARINC 653 specification. Then, Section 3 introduces the main features of the SIGNAL language, while Section 4 concentrates on the modeling of an avionics application in SIGNAL. In section 5, we address issues of performance evaluation for temporal validation based on the SIGNAL language. Finally, conclusions are given in Section 6.

2 The standard ARINC 653

The ARINC specification 653 [4] defines the interface between the application software and the core software (OS, system specific functions), called APEX (APplication EXecutive). This specification is based on the Integrated Modular Avionics approach (IMA) [3]. In an IMA system, several avionics applications which constitute a core module, can be hosted on a single shared computer system. So, one critical aspect is to ensure that shared computer resources are safely allocated so that no fault propagation occurs from one hosted avionics function to another. This is addressed using a *partitioning* of the system. Basically, it consists in a functional decomposition of the avionics applications, with respect to available time and memory resources.
A *partition* [4] is an allocation unit resulting from this decomposition. Suitable mechanisms are provided in order to prevent a partition from having "abnormal" access to the memory area of another partition. The processor is allocated to each partition for a fixed time window within a major time frame maintained by the core module level OS. A partition cannot be distributed over multiple processors neither in the same core module nor in different core modules. Partitions communicate asynchronously via logical *ports* and *channels*.
Every partition is composed of one or more *processes* which represent the executive

[1] There is also an industrial version, SILDEX, implemented and commercialized by TNI-Valiosys (`http://www.tni-valiosys.com`).

units[2]. Processes run concurrently to achieve functions associated with the partition they are contained in. Each process is uniquely characterized by information (priority, deadline time, etc.), 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. The communications between processes are achieved by three basic mechanisms. The bounded *buffer* is used to send and receive messages. It allows storing messages in FIFO queues. The *event* permits the application to notify an occurrence of a condition to processes which may wait for it. The *blackboard* is used to display and read messages; no message queues are allowed, and any message written in a blackboard remains there until the message is either cleared or overwritten by a new instance of the message. Synchronizations are achieved by a *semaphore*.

The APEX interface includes services for communication between partitions on the one hand, and processes on the other hand; services for the synchronization of processes; services for management of partitions and processes, etc.

3 About the SIGNAL language

The underlying theory of the synchronous approach [1] is that of discrete event systems and automata theory. Time is logical: it is handled according to partial order and simultaneity of events. Durations of execution are viewed as constraints to be verified at the implementation level. Typical examples of synchronous languages are ESTEREL, LUSTRE, or SIGNAL which is used here.

The SIGNAL language [9] handles unbounded series of typed values $(x_t)_{t \in \mathbb{N}}$, denoted as `x` in the language, implicitly indexed by discrete time (denoted by t in the semantic notation): they are called *signals*. At a given instant, a signal may be present, then it holds a value; or absent, then it is denoted by the special symbol $\perp$ in the semantic notation. There is a particular type of signals called `event`. A signal of this type is always *true* when it is present (otherwise, it is $\perp$). The set of instants where a signal `x` is present is called its *clock*. It is noted as `^x` (which is of type `event`) in the language. Signals that have a same clock are said to be *synchronous*. A SIGNAL program, also called *process*, is a system of equations over signals. The SIGNAL language relies on a handful of primitive constructs which are combined using a composition operator. They are:

- **Functions.** `y:= f(x1,...,xn)`, where $y_t \neq \perp \Leftrightarrow x1_t \neq \perp \Leftrightarrow ... \Leftrightarrow xn_t \neq \perp$, and $\forall t$: $y_t = f(x1_t, ..., xn_t)$.
- **Delay.** `y:= x $ 1 init y0`, where $x_t \neq \perp \Leftrightarrow y_t \neq \perp$, $\forall t > 0$: $y_t = x_{t-1}, y_0 = y0$.
- **2-arguments down-sampling.** `y:= x when b`, where $y_t = x_t$ if $b_t = true$, else $y_t = \perp$.
- **Deterministic merging.** `z:= x default y`, where $z_t = x_t$ if $x_t \neq \perp$, else $z_t = y_t$.
- **Hiding.** `P where x` denotes that the signal `x` is local to the process `P`.
- **Synchronous parallel composition** of `P` and `Q`, encoded by `(| P | Q |)`. It corresponds to the union of systems of equations represented by `P` and `Q`.

[2] In fact, there is an analogy between ARINC partitions and UNIX processes on the one hand, and ARINC processes and UNIX tasks on the other hand.

These core constructs are of sufficient expressive power to derive other constructs for comfort and structuring. We mention some of the derived operators used in the next sections:

- **1-argument down-sampling.** `y:= when b`, where $y_t = true$ if $b_t = true$, else $y_t = \perp$.
- **Synchronizer.** `x1 ^= ... ^= xn`, where $x1_t \neq \perp \Leftrightarrow ... \Leftrightarrow xn_t \neq \perp$ (i.e. $x1, ..., xn$ are synchronous).
- **Clock union.** `y := x1 ^+ ... ^+ xn`, where $y_t \neq \perp \Leftrightarrow (x1_t \neq \perp \vee ... \vee xn_t \neq \perp)$.
- **Memorizing.** `y := var x init y0`, where `y` always carries the latest value of `x`. The clock of `y` is defined by the context in which it is used.

Verification and analysis of SIGNAL programs. Two kinds of properties may be distinguished: *invariant* properties (e.g. a program exhibits no contradiction between clocks of involved signals) on the one hand, and *dynamical* properties (e.g. reachability, liveness) on the other hand. The SIGNAL compiler itself addresses only invariant properties. For a given SIGNAL program, it checks the consistency of constraints between clocks of signals, and statically proves properties (e.g. the so-called *endochrony* property guaranteeing determinism). A major part of the compiler task is referred to as the *clock calculus*. Dynamical properties are addressed using other connected tools like SIGALI [14], an associated formal system that can be used for model checking. Performance evaluation is another functionality of POLYCHRONY, Section 5 discusses it in a detailed way.

Finally, put together, all these features of SIGNAL programming favor modular and reliable designs.

4 Modeling of an avionics application

A presentation of the basic component models required for the description of avionics applications has been given in [6]. We discussed the modeling of APEX services (communication and synchronization services, process management services, etc.). On the other hand, a model has been proposed for ARINC processes[3]. Here, we show how the models can be used to describe avionics applications[4]. Moreover, we illustrate how performance evaluation applies on the resulting description, for instance to determine the worst case execution times of applications.

Informal specification of the application. figure 1 depicts the considered application which is represented by a partition called *ON_FLIGHT*. Roughly speaking, it is in charge of computing the current position and the fuel level. A report message is produced in the following format:

`[date_of_the_report::height::latitude::longitude::fuel_level]`

The partition includes the following objects:

[3] We use “ARINC processes” to distinguish from SIGNAL processes which are not identical.

[4] The example considered in the following takes its inspiration from a real world avionics application which is currently being modeled.

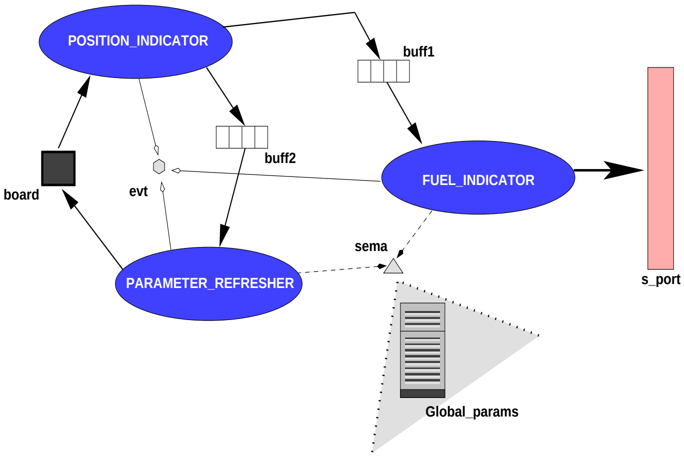

Figure1. The partition ON_FLIGHT.

– A blackboard `board`, two buffers `buff1` and `buff2`, an event `evt`, a semaphore `sema`, and a sampling port[5] `s_port`.
– A resource `Global_params` contains parameters required by the partition processes.
– There are three processes.
  1. The process *POSITION_INDICATOR* first produces the report message which is updated with the current position information (height, latitude and longitude). It works as follows:

     elaborate the report message and set the current date;
     send a request to the process PARAMETER_REFRESHER for a refreshment of global parameters, via buff2 (in order to be able to update the report message with position information);
     wait for notification of end of refreshment, using evt;
     read the refreshed position values displayed on board;
     update the report message with height, latitude and longitude informations;
     send the report message to the process FUEL_INDICATOR, via buff1;
  2. The main task of *FUEL_INDICATOR* is to update the report message (produced by *POSITION_INDICATOR*) with the current fuel level.

     if** a message is contained in the buffer buff1 **then
     retrieve this message;
     end if
     update it with the fuel level information from Global_params, via a protected access (using sema);
     send the final report message via the sampling port s_port;

[5] A sampling port allows no message queuing. There are two kinds of ports: *source* and *destination*. A message remains in a source port until it is transmitted by the channel or it is overwritten by a new occurrence of the message. During transmissions, channels ensure that messages leave source ports and reach destination ports in the same order. A received message remains in the destination port until it is overwritten.

re-initialize evt;

3. Finally, the process *PARAMETER_REFRESHER* refreshes all the global parameters used by the other processes in the partition.

   if** a refresh request arrives in the buffer buff2 **then
       retrieve this message;
   end if
   refresh all the global parameters in Global_params, using a protected access;
   display refreshed position values on board;
   notify the end of the refreshment, using evt;

From the above informal specification, we now derive an associated synchronous model.

The SIGNAL **model of the partition.** The executive model of a partition consists of three basic components: first, the executive units represented by ARINC processes; second, the interactions between processes described by APEX services; and finally, the partition level OS which allocates the resources (e.g. processor, communication mechanisms) to processes within the partition.

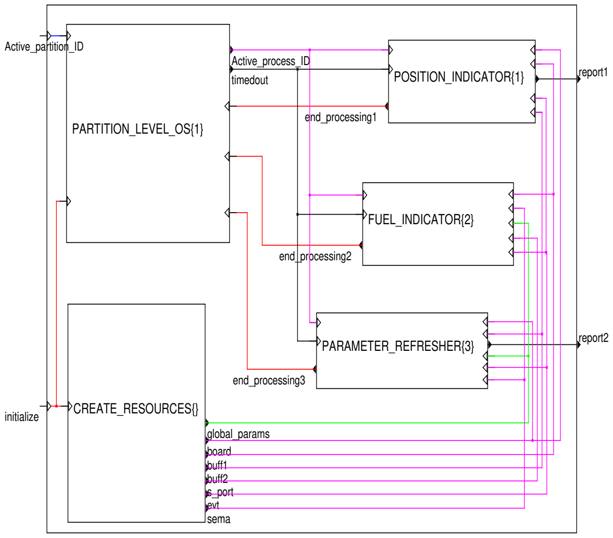

Figure2. A SIGNAL model of the partition ON_FLIGHT.

figure 2 depicts the model corresponding to the partition *ON_FLIGHT*, obtained with POLYCHRONY. We clearly distinguish the partition level OS as well as the three processes. The box that contains the SIGNAL process `CREATE_RESOURCES` has been added for structuring. It provides the processes with communication and synchronization mechanisms (e.g. `buff1`, `sema`). These mechanisms are created on the occurrence of the input signal `initialize`. The presence of this signal corresponds to the *initialization* phase of the partition. The input `Active_partition_ID` represents the

identifier of the running partition selected by the module level OS[6], and it denotes an execution order when it identifies the current partition. Whenever the partition executes, the partition level OS designates an active process within the partition. This is represented by its output signal `Active_process_ID`. It is sent to all the processes. Every process which runs to completion notifies the OS through a special signal (e.g. `end_processing1` for the ARINC process `POSITION_INDICATOR`), in order to take a decision about the next process to execute.
A process can be blocked during its execution. For instance, when it tries to send a message to a full buffer. So, a time counter may be initiated to wait for the availability of space in the buffer. The signal `timedout` produced by the partition level OS notifies processes the expiration of their associated time counters.

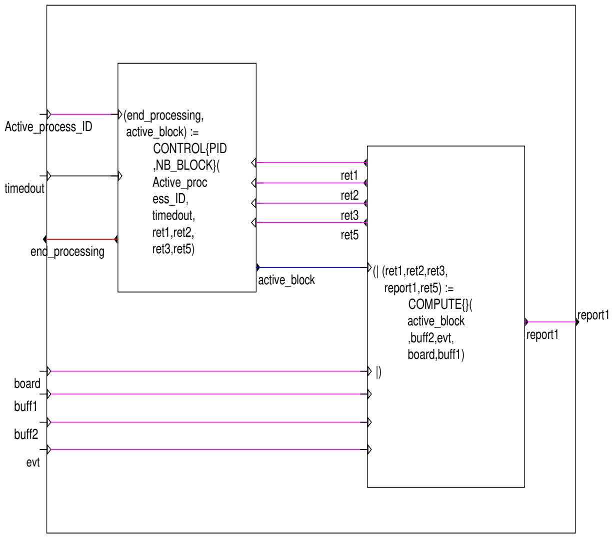

Figure3. A SIGNAL model of the process POSITION_INDICATOR.

Modeling of processes. To illustrate the description of the processes in *ON_FLIGHT*, we mainly focus on the process *POSITION_INDICATOR* (the modeling of the other processes follows the same scheme).
As shown in **figure** 3, a process is basically composed of two sub-components: *CONTROL* and *COMPUTE*. The former specifies the execution flow of an ARINC process. Typically, it is a transition system that indicates which statements in the body of the process should be executed whenever the process is active. The latter describes the actions computed by a process. It is composed of so-called *Blocks*. They represent elementary

[6] 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.

pieces of code to be executed without interruption like *Filaments* [5]. Furthermore, the statements associated with a Block must complete within a bounded amount of time.

In **figure** 3, the signal `active_block` identifies a Block selected by the *CONTROL* sub-component. This Block is executed instantaneously. Therefore, one must take care of what kinds of statements can be put together in a same Block. Two sorts of statements can be distinguished. First, those which may cause an interruption of the running process (e.g. a *SEND_BUFFER* request on a full buffer). We call them *system calls* (in reference to the fact that they involve the partition level OS). The others are statements that never interrupt the running process. Typically, data computation functions. They are referred to as *functions*. Clearly, only one system call at most can be associated with a Block, and no other statement can follow this system call within the Block. Since a Block is executed instantaneously, what would happen if the system call interrupts the running process? All the other statements within the Block would be executed in spite of the interruption, and this would not be correct. Moreover, when the process is resumed, the whole Block may not necessarily require to be re-executed, so one must take care of that.

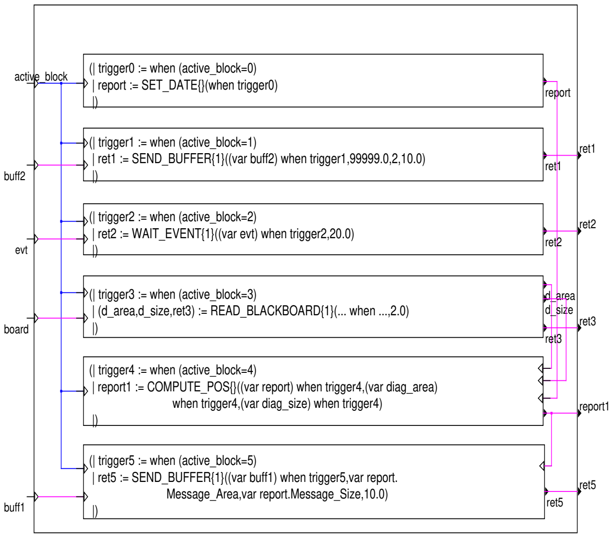

Figure4. COMPUTE sub-component of the process POSITION_INDICATOR.

The *COMPUTE* sub-component is depicted in **figure** 4. The Blocks (represented by boxes within the model) describe the actions that are achieved by *POSITION_INDICATOR*. The statements associated with a Block k are executed whenever the Block is selected by the *CONTROL* sub-component, i.e. whenever the event `triggerk` is present. For instance, from top to bottom, the first Block contains a function `SET_DATE` which produces an instance of the report message, where only the field `date_of_the_report` is updated. The other fields will be completed later. The second Block contains the system call `SEND_BUFFER`, which is used to send a message in the buffer `buff2`. Input parameters are the message address and size (respectively, denoted by 99999.0 and 2), and a time-out value (10.0 time units) to wait for space when the buffer is full. A return code (`ret1`) is sent for diagnostic. Here, Blocks are computed sequentially from top to bottom. However, there could be consecutive executions of a same Block. This happens when a system call is executed and the required resource is not yet available. For example, consider the `READ_BLACKBOARD` request in the fourth Block from the top (used to get a message from `board`), if no message is currently displayed in the blackboard, the calling process will get suspended on this Block. After the availability of a message, the process is put in the "ready" state. As soon as it becomes active, it should re-execute the same Block (which induced its suspension) to read the latest message available in the blackboard. The automaton which describes the execution of Blocks is specified within the *CONTROL* sub-component. Automata are very easy to specify in SIGNAL.

Modeling of the partition level OS. The main task of the partition level OS is to ensure a correct concurrent execution of processes within the partition. Its modeling requires on the one hand, APEX services (e.g. in **figure** 5, `CREATE_PROCESS` and `START` used respectively to create and start processes), and implementation-dependent functions, for instance to define a scheduling policy (e.g. `PROCESS_SCHEDULINGREQUEST` in **figure** 5) on the other hand.

```
(| (att1,att2,att3) := GET_PROCESSES_ATTRIBUTES{}(when initialize)        (a)
 | (pid1,return_code1) := CREATE_PROCESS{}(att1 when initialize)          (b)
 | (pid2,return_code2) := CREATE_PROCESS{}(att2 when initialize)
 | (pid3,return_code3) := CREATE_PROCESS{}(att3 when initialize)
 | return_code4 := SET_PARTITION_MODE{}(#NORMAL when (^return_code3))     (c)
 | return_code5 := START{}(pid1)                                          (d)
 | return_code6 := START{}(pid2)
 | return_code7 := START{}(pid3)
 | partition_is_running := (Active_partition_ID = Partition_ID)           (e)
 | diagnostic := PROCESS_SCHEDULINGREQUEST{}(
                           when partition_is_running)                     (f)
 | (Active_process_ID,status) := PROCESS_GETACTIVESTATUS{}()              (g)
 | timedout := UPDATE_COUNTERS{}()                                        (h)
 | Active_process_ID ^= timedout ^= when partition_is_running
 | return_code8 := SUSPEND{}(Active_process_ID when (end_processing1
                        ^+ end_processing2 ^+ end_processing3))           (i)
 | return_code9 := SET_PARTITION_MODE{}(#IDLE when (^end_processing2))    (j)
 |)
```

Figure5. The partition level OS model.

figure 5 shows a partial view of the SIGNAL description of the partition level OS. Let us take a look at the specified equations. On the presence of the signal `initialize` (which corresponds to the initialization phase of the partition), process attributes are first defined in equation `(a)`, example of attributes are process *name*, *priority*, *periodi-*

city. Just after that, processes are created and started[7]. For instance, the lines (b) and (d) correspond to the creation and starting of the process identified by pid1 (in fact POSITION_INDICATOR). In the equation (c), the partition is set to the *NORMAL* mode[8]. The signal Active_partition_ID represents the identifier of the running partition selected by the *module level* OS. It denotes an execution order when it identifies the current partition, this is the meaning of the boolean partition_is_running definition in (e). So, process rescheduling is performed whenever the partition is active, and the process with the highest priority in the ready state is designated to execute. The process is identified by Active_process_ID. This is achieved in the equations (f) and (g). On the other hand, all time counters used in the partition are updated whenever the partition executes (equation (h)). The signal timedout is sent to processes to notify them a (possible) expiration of their associated time counters. A running process gets suspended as soon as it completes (one of the signals end_processing1, end_processing2, or end_processing3 is received from the three processes in the partition). This is expressed in equation (i). Finally, the partition is set to *IDLE* mode when no process executes while the partition is still active (line (j)). In the partition, the process which completes the last is FUEL_INDICATOR, and the signal end_processing2 is received from this process.

We observe that using the services defined in the library [6], combined with the features of SIGNAL programming such as modularity, the basic components (executive units, communication and synchronization mechanisms, and partition level OS) required for the modeling of the application are easily specified.
Now, let us consider the resulting model (depicted in **figure** 2). It is represented by a SIGNAL process for which a simulation C code has been generated in order to execute the partition. Furthermore, various properties can be verified on this program using available tools (compiler functionalities, SIGALI, etc.). In particular, one can address timing issues (e.g. to compute worst case execution times) using the performance evaluation technique implemented in POLYCHRONY.

The next section focuses on this technique, it gives the basic principles for deriving a temporal interpretation of a SIGNAL process for the purpose of studying the real-time behavior of modeled applications.

5 Performance evaluation

A SIGNAL process that models an application is recursively composed of sub-processes, where elementary sub-processes belong to the kernel language, let us call them *atomic*

[7] The *START* service only puts the specified process in the "ready" state, the process does not execute yet!

[8] There are four operating modes [4]: in the *IDLE* mode, the partition is not executing any process within its allocated windows; in the *COLD_START* mode, the partition is executing a cold start initialization; in the *WARM_START* mode, the partition is executing a warm start initialization; and in the *NORMAL* mode, the scheduler is activated, and all the required resources in the partition must have been created before.

nodes. A profiling of such a process substitutes each signal with a new signal representing availability dates $date\_x$ and automatically replaces atomic nodes with their timing model counter-part ("timing" morphism). The resulting time model is composed (by standard synchronous composition) with the original functional description of the application, and for each signal x, a synchronization with the signal $date\_x$ is added. The resulting process is close to (or even represents exactly) the model of the temporal behavior of the application running on its actual architecture. One can obviously design less strict modeling to get faster simulation (or formal verification); it is sufficient to consider more abstract representations either of the architecture or of the program.

5.1 Temporal interpretation of SIGNAL processes

An interpretation of a SIGNAL specification is a SIGNAL process that exposes a different view of the initial SIGNAL specification. The structure of the interpretation process is essentially the same but its computations exhibit another aspect of its behavior. The temporal interpretation exposes the time aspect and permits to see how an implementation of a specified function will behave over time [12].

For each SIGNAL process independent of its complexity level, another SIGNAL process can be automatically derived to model its temporal behavior on a given implementation. These processes are called temporal interpretations. For a SIGNAL process P its temporal interpretation for an implementation I will be denoted by $T(P_I)$, where P_I is the SIGNAL process that models implementation I of P. Thus, if a system specified by a SIGNAL process P has a variety of possible implementations $I(1)$ to $I(k)$, then each implementation can be 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.

In this way a comparative performance evaluation of the different implementations can be performed and the design space of possible implementations can be effectively explored before committing the design to one in particular. Such an approach permits to concentrate the design effort to a set of candidate implementations.

Signal availability dates. For each signal in the initial SIGNAL specification a date signal is defined in its temporal interpretation: $x \in P \rightarrow T(x) \in T(P)$.
For any signal x in P we have a $date\_x$ in $T(P)$ with x synchronous to $date\_x$:

$$P \rightarrow T(P),\ x \rightarrow T(x) = date\_x,\ x \hat{=} date\_x.$$

These date signals are some sorts of time-stamps providing the availability times for the values of the corresponding signals in the functional specification, in respect to a global time reference. Depending on the implementation context, time can be measured using either physical time units or full clock cycles. In the first case the date signals are positive real numbers and in the second positive integers. From a cycle count integer measurement we can go on to physical time measurement by multiplying the cycle count to the cycle period.

Each operation in a SIGNAL specification is represented by a node in the Hierarchical Conditional Dependency Graph, which is the internal representation of a SIGNAL program. To each node in the graph, a delay is associated. This delay is represented by the same data type as the data type used to represent dates and is a function of several parameters. The actual node delay is obtained by giving values to these parameters. The delay depends on parameters like: operation performed by the node, data types involved, chosen implementation, etc. Furthermore, a delay can be represented by a pair of numbers corresponding to the worst and best case delays. Having delays represented by intervals results in dates represented as intervals too. Computing these dates takes into account the processing delays.

It is important to underline that this date mechanism permits us to pass from logical to physical time.

Non-functional interpretations. The temporal interpretation of a SIGNAL specification is just a special case of a general non-functional interpretation. The non-functional interpretations are SIGNAL processes and as such they can be decomposed into a control and a data part. The control computations are identical to those in the initial processes from which the interpretations are derived. What changes are the data computations since they extract the information related to the particular interpretation.

For a SIGNAL process P we know that $P = C_P|D_P$, with C_P the control part of the process P and D_P its data part. Similarly an interpretation of P, denoted by $T(P)$, decomposes into a control and a data part: $T(P) = C_{T(P)}|D_{T(P)}$.
Since the interpretation of a complex process can be defined as the recursive composition of the interpretations of the constituent processes for $T(P)$ we have:
$T(P) = T(C_P)|T(D_P)$, with $T(C_P) = C_{T(C_P)}|D_{T(C_P)}$ and $T(D_P) = C_{T(D_P)}|D_{T(D_P)}$.
For the control part, we have $C_{T(P)} = C_P$.

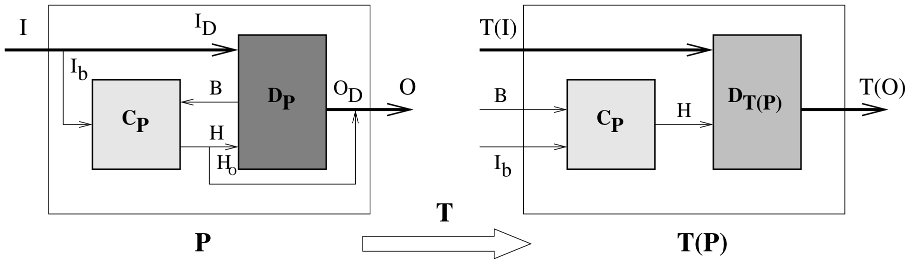

Figure6. Temporal interpretation of a SIGNAL process P.

The process of obtaining an interpretation $T(P)$ of a process P is graphically depicted in **figure** 6. This process gives a general form of *morphism* of SIGNAL programs, which is available in POLYCHRONY. 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, certain outputs are output events (H_O) computed by C_P. The control parts of the initial process and its interpretation are identical, but the data computations differ. The data computations in $T(P)$ extract the information of interest, implicit in the initial specification P.

The date computation model. The SIGNAL kernel operators are the simplest processes that can be used to build more complex ones. Similarly, the interpretation of a process can be viewed as the composition of the interpretations of the primitive processes making up the initial process.

The interpretations of the kernel processes perform the appropriate computations relating to a particular interpretation. These interpretations are organized in a collection which represents the *library of cost functions*, defined in SIGNAL. For each interpreted process, this library is extended with the interpretations of external function calls and other separately compiled processes, used in the initial process. For example, the "timing" morphism available in POLYCHRONY associates with the monochronous addition operator `z := x + y`, the following process:

$$\texttt{(date\_z,done}_i\texttt{) := CostPlus}\{\texttt{type(x), type(y)}\}$$
$$\texttt{(date\_x, date\_y, date\_clk\_z, wait}_i\texttt{)}$$

The signal `date_clk_z` is the signal associated with the common clock of `x`, `y` and `z` by the morphism, the notation `type(x)` represents the type of `x`. Signals $\texttt{wait}_i$ and $\texttt{done}_i$ are associated with the current node and have the same type as date signals: $\texttt{wait}_i$ accumulates dates coming from incoming precedences other than data dependencies, $\texttt{done}_i$ is defined as being, roughly, $\texttt{date\_z default wait}_i$.
The first output of the cost function `CostPlus`, which corresponds to the date of the result, is defined as being the sum of the delay taken to perform the addition operation (some Δ_+) and the maximum of the dates corresponding to the inputs of `CostPlus`. The 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 architecture. In the current implementation in POLYCHRONY, the value Δ_+ is provided by a function `getCostPlus` which has the types of the operands as parameters and which fetches the required value in some table.

The scheme we have illustrated for monochronous operators handles also "control" operators. For constructs such as the `default` operator, which allows for control branching, the definition of the associated interpretation accounts for this branching (for `a default b`, the date at which the input value is available is given by `date_a default date_b`).

Moreover, thanks to compositionality of SIGNAL specifications, the above mechanism can be applied at any level of granularity.

5.2 Obtaining results

figure 7 depicts a *co-simulation* of the application model composed with its associated temporal interpretation. At each iteration, the date of an output ($d(O_k)$) depends on the date of an input ($d(I_j)$) and the *control configuration* represented by a "valuation" of a condition vector $[c_1, \ldots, c_q]$ corresponding to intermediate boolean signals B (cf. Fig. 6) computed in the original program. In a straightforward approach, it is possible to provide a set of vectors that covers all the possible combinations for the control flow. A better possibility is to take into account the existing relationships between these booleans such as provided by the clock calculus (this is expressed through the composition of the original program and its temporal interpretation). In addition, specific *observer* processes, comparing dates or verifying some conditions (timing requirements) for example, can be inserted into the model.

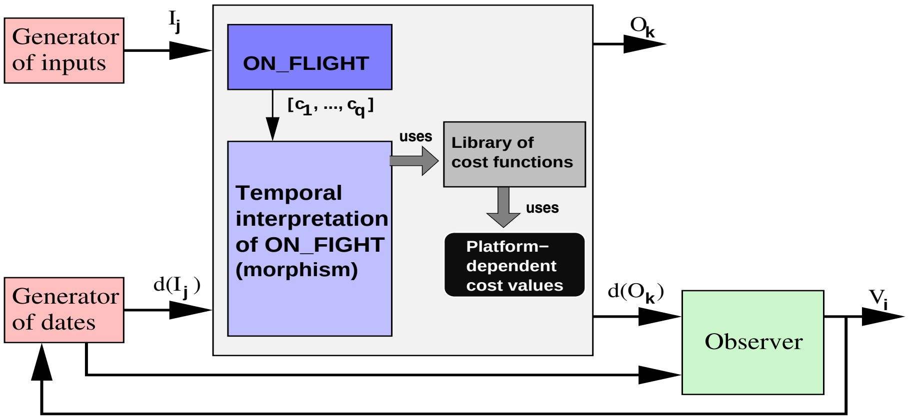

Figure7. Co-simulation of the application with its temporal interpretation.

Finally, all the tools available in the synchronous technology can be used on such a model, including for example some formal verifications when the corresponding required abstractions are considered.

Several successful experiments have been done on sample SIGNAL programs. In the case of the ON_FLIGHT model, some simplifications have been made because of the complexity of used data structures. So, the cost of the accesses to those data structures and related effects is not taken into account. The cost function library currently takes into account only (relatively) simple data structures (e.g. integer, boolean, arrays). The others are considered as external. As a result, the current computed results are not enough relevant to be highlighted here. However, the cost function library is being currently enhanced to allow more efficient experiments on programs with complex data structures. Thus, more relevant results will be available very soon.

6 Conclusions

In this paper, we have illustrated a component-based approach to the modeling of avionics applications for the purpose of formal verification and analysis. The whole approach relies on the use of a single formalism of the SIGNAL language. This is part of a more general design methodology for distributed embedded application, defined within POLYCHRONY. This methodology proceeds by successive transformations on an initial SIGNAL model that preserve semantic properties. During the transformations, "abstract" components can be instantiated in different ways from modules related to actual target architecture features, addressing various purposes (embedded code generation, temporal validation, etc.). Here, we considered models of APEX services [6] to describe avionics applications. Then, we used the POLYCHRONY tool-set to analyze applications, in particular, we focused on the real-time behavior. The technique [12] we considered is still being implemented to take into account SIGNAL programs with complex data structures (such as the model of the partition ON_FLIGHT described in this paper).

References

1. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Proceeding of the IEEE, vol. 79, No. 9*, pages 1270–1282, April 1991.
2. E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys: a tool for the development and verification of real-time embedded systems. In *Proceedings of Computer Aided Verification, CAV'01. Paris, France. Lecture Notes in Computer Science 2102, Springer-Verlag*, July 2001.
3. Airlines Electronic Engineering Committee. Arinc report 651-1: Design guidance for integrated modular avionics. In *Aeronautical radio, Inc., Annapolis, Maryland*, November 1997.
4. Airlines Electronic Engineering Committee. Arinc specification 653: Avionics application software standard interface. In *Aeronautical radio, Inc., Annapolis, Maryland*, January 1997.
5. D. Engler, D. Andrews, and D. Lowenthal. Efficient support for fine-grain parallelism. In *Technical report, University of Arizona*, 1993.
6. A. Gamatié and T. Gautier. Modeling of modular avionics architectures using the synchronous language SIGNAL. In *Proceedings of the Work In Progress session, 14th Euromicro Conference on Real Time Systems, ECRTS'02*, pages 25–28. Vienna, Austria, June 2002. Complete version is available as INRIA research report n. 4678, December 2002.
7. A. Gamatié and T. Gautier. The SIGNAL approach to the design of system architectures. In *Proc. of the 10th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'03), IEEE Computer Society Press*, 2003. (To appear).
8. T. Gautier and P. Le Guernic. Code generation in the sacres project. In *Proceedings of the Safety-critical Systems Symposium, SSS'99, Springer.* Huntingdon, UK, February 1999.
9. T. Gautier, P. Le Guernic, and O. Maffeïs. For a new real-time methodology. In *the INRIA Research Report number 2364 (`http://www.inria.fr/rrrt/rr-2364.html`)*, October 1994.
10. D. Goshen-Meskin, V. Gafni, and M. Winokur. SAFEAIR: An integrated development environment and methodology. In *INCOSE 2001, Melbourne*, July 2001.
11. T. A. Henzinger, B. Horowitz, and Ch. Meyer Kirsch. Embedded control systems development with giotto. In *Proceedings of LCTES. ACM SIGPLAN Notices*, 2001.
12. A. Kountouris and P. Le Guernic. Profiling of SIGNAL programs and its application in the timing evaluation of design implementations. In *Proc. of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems, IEE*, pages 6/1–6/9. HP Labs, Bristol, UK, February 1996.
13. E. A. Lee and al. Overview of the ptolemy project. In *Technical Report UBC/ERL M01/11, University of California at Berkeley*, March 2001.
14. H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. In *Discrete Event Dynamic System: Theory and Applications, 10(4)*, pages 325–346, October 2000.
15. J. Sifakis. Modeling real-time systems - challenges and work directions. In *EMSOFT'01, Tahoe City. Lecture Notes in Computer Science 2211*, October 2001.
16. S. Vestal. Metah support for real-time multi-processor avionics. In *IEEE Workshop on Parallel and Distributed Real-Time Systems*, April 1997.