Synchronous Modeling of Avionics Applications using the SIGNAL Language*

Abdoulaye GAMATIÉ, Thierry GAUTIER
IRISA / INRIA
F-35042 RENNES, France
{agamatie, gautier}@irisa.fr

Abstract

In this paper, we discuss a synchronous, component-based approach to the modeling of avionics applications. The specification of the components relies on the avionics standard ARINC 653 and the synchronous language SIGNAL *is considered as modeling formalism. The* POLYCHRONY *tool-set allows for a seamless design process based on the* SIGNAL *model, which provides possibilities of high level specifications, verification and analysis of the specifications at very early stages of the design, and finally automatic code generation through formal transformations of these specifications. This suits the basic stringent requirements that should be met by any design environment for embedded applications in general, and avionics applications in particular.*

1. Introduction

Embedded systems are an integral part of safety critical systems encountered in various domains, such as avionics, automotive, telecommunications. Among the challenges in the design of systems for avionics applications, we first mention the correctness of the design with respect to the requirements; then, the development effort and time to market; and finally, the correctness and reliability of the implementation. A major objective of the SAFEAIR project approach (`http://www.safeair.org/`) is to improve the embedded systems development process, allowing to maintain the high level of dependability of aircraft systems in the face of an exponential growth in functionality and complexity. The work presented in this paper aims to contribute to the SAFEAIR solution.

Today, it is widely accepted that modeling is essential to the design activity for embedded systems. It enables experiments without having necessarily the actual system. So, more flexibility is allowed when taking decisions because one can do more simulation in short time. Other advantages already emphasized by [17] are genericity, abstraction, non determinism, and formal methods for analysis and predictability.

Several *model-based* approaches have been proposed [18, 10, 13, 3] 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. Our approach aims at the same objective. Its main particularity relies on the use of a single semantical model, SIGNAL [2], to describe embedded applications from specification to implementation. Verification and analysis of specifications are enabled by means of well-defined formal transformations that preserve the semantics of initial specifications. This favors the validation. In other words, the approach considers basic challenges in specification and verification of embedded systems [15].

Over the past decade, the synchronous technology [9] has emerged as one of the most promising ways for guaranteeing a safe design of embedded systems. It offers practical design assistance tools with a formal basis that allow high level specifications, verification and analysis for validation, and automatic code generation. POLYCHRONY, the tool-set of SIGNAL developed by INRIA[1] (`http://www.irisa.fr/espresso/Polychrony`), includes all these functionalities.

In the rest of the paper, we mainly focus on the definition of SIGNAL models of components required for the description of avionics applications. A previous introduction to this approach has been presented in [7]. Section 2 discusses two design concepts for avionics systems: the *federated* and *Modular Integrated* approaches. The standard ARINC 653 specification [4] is based on the latter. Section 3 presents the main features of SIGNAL, while section 4 concentrates on the modeling of the ARINC specification with SIGNAL. In section 5, we relate our approach to the literature. Finally, conclusions are given in section 6.

*This work has been supported by the european project IST SAFEAIR (Advanced Design Tools for Aircraft Systems and Airborne Software) [8].

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

2. Avionics architectures

Avionics architecture designs adopt more and more the *integrated modular* solution rather than the traditional *federated* one [16]. As a matter of fact, in federated architectures there is a potential risk of massive use of resources since each avionics function requires its own computer system, most of the time replicated for fault tolerance. However, a great advantage to these architectures is fault containment. The Integrated Modular Avionics approach (I-MA) [5] allows to face the problem of resource usage that exists in the federated approach.

In IMA architectures, several avionics functions can be hosted on a single, shared computer system. Therefore, a critical aspect is to ensure that shared computer resources are safely allocated so that no fault propagation occurs from one hosted function to another. This is addressed by the *partitioning* mechanism. It consists in a functional decomposition of avionics applications, with respect to available time and memory resources. A *partition* [4] is an allocation unit resulting from this decomposition. Partitioning promotes verification, validation, and certification.

Partitions. A *core module* encompasses several partitions that possibly belong to applications of different critical levels. 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 either in the same module or in different modules. Finally, partitions communicate asynchronously via logical *ports* and *channels*.

Processes. Partitions are composed of *processes* that represent the executive units[2]. Processes run concurrently and execute functions associated with the partition they are contained in. Each process is uniquely characterized by informations (like its period, priority, or deadline time), useful to the partition level OS which is responsible for the correct execution of processes within a partition. The scheduling policy for processes is priority preemptive. Communications between processes are achieved by three basic mechanisms. The bounded *buffer* allows to send and receive messages following a FIFO policy. The *event* permits the application to notify processes of the occurrence of a condition for which they may be waiting. The *blackboard* is used to display and read messages; no message queues are allowed, and any message written in 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*.

The ARINC 653 specification [4] relies on IMA. It defines the interface between the application software and the core software (OS, system specific functions), called APEX (APplication EXecutive). It includes services for communication between partitions on the one hand and processes on the other hand, synchronization services for processes, partition and process management services, etc.

[2] An ARINC partition/process is akin a UNIX process/tasks.

3. The synchronous language SIGNAL

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 [9] are: ESTEREL, LUSTRE or SIGNAL. They mainly differ from each other in their programming style. ESTEREL adopts an imperative style whereas the two others are data-flow oriented (LUSTRE is functional and SIGNAL is relational).

Main characteristics of the language. SIGNAL [2] 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) and 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. 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 the same clock are said to be *synchronous*. A SIGNAL program, also called *process*, is a system of equations over signals. The SIGNAL language [12] relies on a handful of primitive constructs which are combined using a composition operator. These core constructs are of sufficient expressive power to derive other constructs for comfort and structuring.

SIGNAL also provides a process frame in which any process may be "encapsulated". This allows to abstract a process to an interface, so that the process can be used afterwards as a black box through its interface which describes the input-output signals and parameters. The process frame enables the definition of sub-processes. Sub-processes which are only specified by an interface without internal behavior are considered as external (they may be separately compiled processes or physical components). On the other hand, SIGNAL allows to import external modules (e.g. C++ functions). Finally, put together, all these features of the language favor modularity and reusability.

Verification and performance evaluation. Two kinds of properties may be distinguished: *invariant* properties (e.g. a program exhibits no contradiction between clocks of involved signals), and *dynamical* properties (e.g. reachability, liveness). 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. Basically, it consists of formal transformations (*morphisms*) of a SIGNAL program [11] describing an application, which yield another SIGNAL program that corresponds to a temporal interpretation of the initial program. A co-simulation of the resulting program together with the original one can provide, for example, worst case execution times.

In the next section, we show how components required for the modeling of avionics applications are specified with the SIGNAL language. The ARINC 653 specification is considered as basis for the models.

4. Modeling of components for the description of avionics applications

Before presenting the modeling of each component, let us observe that the executable model of a partition consists of three basic components. First, executive units which are represented by *ARINC processes*[3]. Second, the interactions between processes described by *APEX services*. Finally, the *partition level OS* which is in charge of the correct access to resources (e.g. processor) by processes within the partition. So, we first focus on the description of APEX services in sub-section 4.1; then, a model of ARINC processes is proposed in sub-section 4.2; and finally, we discuss the modeling of the partition level OS in sub-section 4.3.

4.1. APEX services

APEX_services include communication and synchronization services used by processes (e.g. *SEND_BUFFER*, *WAIT_EVENT*, *READ_BLACKBOARD*), process management services (e.g. *START*, *RESUME*), partition management services (e.g. *SET_PARTITION_MODE*), and time management services (e.g. *PERIODIC_WAIT*). The modeling approach is illustrated with the help of an APEX service. We show how the corresponding SIGNAL model is obtained from informal specifications like those encountered in [4].

Modeling of an APEX service. Let us consider the *READ_BLACKBOARD* service [4], used to read a message in a blackboard. The input parameters are the blackboard *identifier*, and a *time-out* duration that limits the waiting time if the blackboard is empty. The outputs are a *message* (defined by its address[4] and size), and a *return code* for the diagnostics of the service request. An informal specification is as follows:

if *inputs are invalid (that means the blackboard identifier is unknown or the time-out value is "out of range")* ***then***
    return *INVALID_PARAM;*
else if *some 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;
    end if
    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;*
    end if
end if

Analysis of the problem. To understand how to derive a corresponding synchronous model from the service, let us consider the concurrent execution of two processes P1 and P2 within a partition. The process P1 is assumed to have a higher priority than P2. They communicate via a blackboard which is currently empty. Two possible scenarios are illustrated in **Figure 1**.
In both scenarios, P1 tries to read the blackboard before P2, and gets suspended since no message is displayed yet. As a result, a re-scheduling is performed, and P2 runs. The process P1 must wait for either a notification that an initiated time counter became zero[5] (*situation A*), or the availability of some message (displayed by P2) in the blackboard (*situation B*). Now, if we check the time-line in both situations, we observe that the time-lag corresponding to the read_blackboard service is `[t2, t3]`. It partially includes the executions of P1 and P2. We remind that within a partition, only one process executes at any instant. In a synchronous view, it can be interpreted as: *only the statements associated with one process are executed within any synchronous step*. Clearly, we have to split the service into subsets of actions since the whole service cannot be entirely executed within a single synchronous step. Therefore, we distinguish two subsets: on the one hand, actions executed

[3] We use the terms "ARINC processes" to distinguish from SIGNAL processes which are not identical.

[4] Also referred to as *area*.

[5] In the informal specification, it corresponds to the emission of *TIMED_OUT* as return code value.

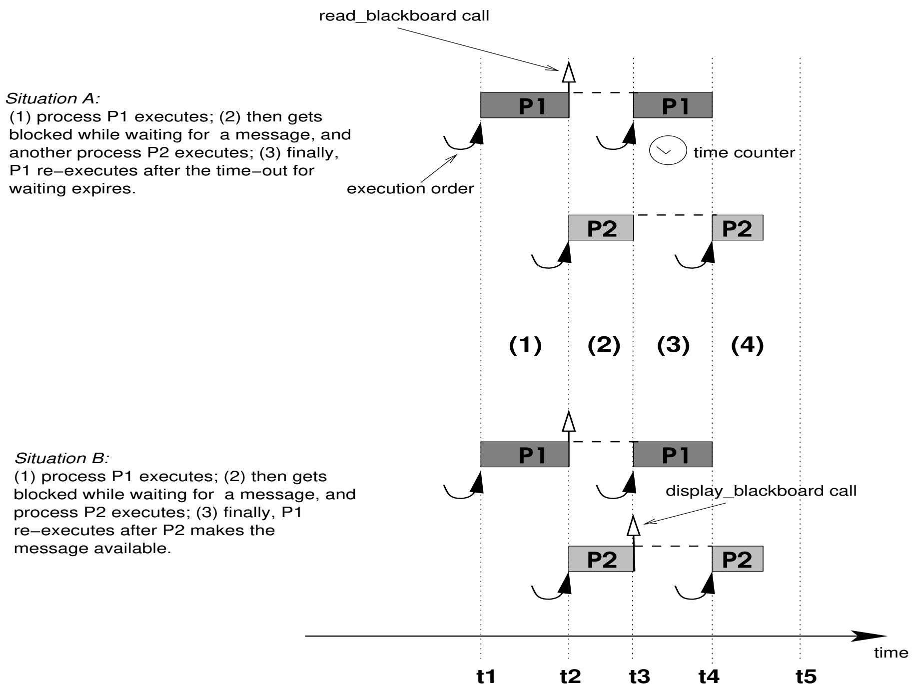

Figure 1. Concurrent execution of two processes P1 and P2 on one processor.

when P1 is running (e.g. checking the validity of input parameters or initiating a time counter), we call them *local actions*; and actions performed during its suspension (e.g. in *situation A*, these actions consist of the control of the time counter: decrease it and notify when it becomes zero), referred to as *global actions*.

In fact, global actions will be under the control of the partition level OS, which is responsible for the management of all processes, shared resources and mechanisms within the partition. Sub-section 4.3 discusses the modeling of the partition level OS.

The SIGNAL **model.** Now, we show how the local actions of the READ_BLACKBOARD service are modeled with SIGNAL. For that, we consider the *situation B* of **Figure 1** where P1 resumes after P2 has displayed a message on the blackboard. Local actions induced by READ_BLACKBOARD call take place exactly at `t2` (e.g. check the validity of input parameters or initiate a time counter for waiting) and `t3` (e.g. retrieve the latest available message). Let `L` and `L'` denote the respective subsets of local actions that occur at these instants. They are grouped into the same SIGNAL process which represents a partial model of the service. On the other hand, since they are not achieved at the same point in time, we have to define the conditions which select the right subset of local actions to be executed whenever the whole SIGNAL model is activated. This is easily described using an internal state variable that indicates which one among `L` and `L'` should be computed. Typically, it is encoded by a boolean signal `blocked` (that initially carries the value *false*) as depicted in **Figure 2**.

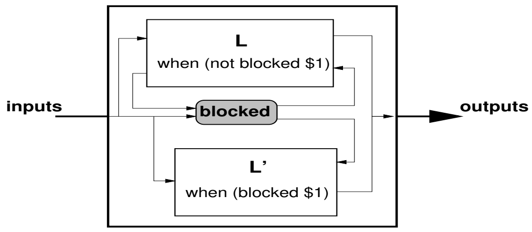

Figure 2. Rough model of local actions associated with a blocking service.

In this model, `L` is executed when the caller was not previously blocked on the service call (denoted by the condition `when (not blocked $ 1)` in the figure). The boolean `blocked` is set to *true* as soon as the resource is not available (empty blackboard). This information is represented by the arrow from `L` to `blocked` in the figure. When the state variable previously carried the value *true* (i.e. the caller has been blocked previously), the subset `L'` is executed and the boolean `blocked` becomes *false*.

The SIGNAL process[6] shown in **Figure 3** models the

[6]Note that we call it READ_BLACKBOARD even though it only des-

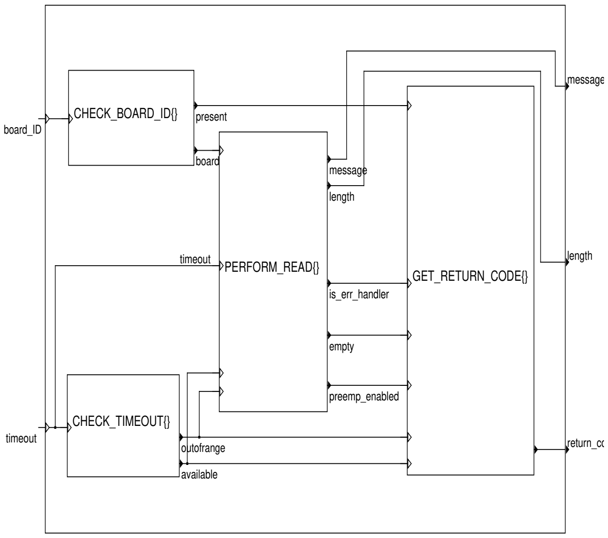

Figure 3. SIGNAL model of the READ_BLACKBOARD service.

local actions executed on a READ_BLACK-BOARD request. There are four main sub-processes. The sub-processes `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. Afterwards, it sends the latest message displayed on the blackboard (its area and size are specified by `message` and `length`). It also transmits all the necessary informations to `GET_RETURN_CODE` which defines the final diagnostic message of the service request. For example, when signals `empty` and `preemp_enabled` respectively carry the values `true` and `false`, `GET_RETURN_CODE` sends `INVALID_MODE` as `return_code` (that means the service caller is suspended until a message becomes available, and no other process can execute during the suspension because preemption is not enabled in the current operating mode). In the case of invalid inputs (e.g. `board_ID` is an unknown identifier within the partition, or `timeout` is "out of range"), informations are still sent to `GET_RETURN_CODE` by `CHECK_BOARD_ID` and `CHECK_TIMEOUT` in order to determine the return code.
The modeling of the other APEX services follows the same scheme.

4.2. ARINC process

The definition of an ARINC process model basically takes into account the computation and control parts of an ARINC process. This is depicted in **Figure 4**. Two

cribes the local actions.

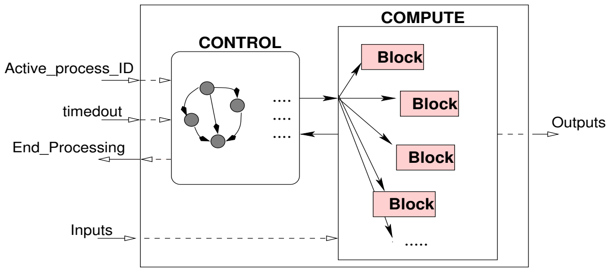

Figure 4. ARINC process model.

sub-components are clearly distinguished within the model: *CONTROL* and *COMPUTE*. Any ARINC process is seen as a reactive component, that reacts whenever an execution order (denoted by the input `Active_process_ID`) is received. The input `timedout` notifies processes of time-out expiration, while the output `End_Processing` is emitted by the process after completion. In addition, there are other inputs (resp. outputs) needed for (resp. produced by) the process computations. The CONTROL and COMPUTE sub-components cooperate to achieve the correct execution of the process model.
The CONTROL sub-component. It specifies the control part of the ARINC process. Basically, it is a transition system that indicates which statements should be executed whenever 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 ARINC process, this process "executes". Depending on the current state of the transition system representing the execution flow of the process, a *Block* of actions in the COMPUTE sub-component is selected to be executed *instantaneously* (this is represented by the arrow from CONTROL to COMPUTE in the figure).
The COMPUTE sub-component. It 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 like *Filaments* [6]. The statements associated with a Block are assumed to *complete within a bounded amount of time*. In the model, a Block is executed instantaneously. Therefore, one must take care of what 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*.
For a correct execution, only one system call at most must be associated with a Block, and no other statement should follow this system call within the Block. As a matter of fact,

a Block is executed instantaneously, so 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. Furthermore when the process gets resumed, the whole Block may not necessarily require to be re-executed.

The process model proposed here is very simple. An execution of ARINC processes can be seen as a sequence of Blocks, and preemption is represented by an occurrence of two consecutive Blocks that belong to different processes in a sequence.

4.3. 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 as follows:

```
process PARTITION_LEVEL_OS =
 { PartitionID_type Partition_ID; }
 ( ? PartitionID_type active_partition_ID;
     event initialize;
     event end_processing;
   ! ProcessID_type active_process_ID;
     [3]boolean timedout;
 )
(| (pid1,ret_c1):= CREATE_PROCESS(att1 when initialize)
 | (pid2,ret_c2):= CREATE_PROCESS(att2 when initialize)
 | (pid3,ret_c3):= CREATE_PROCESS(att3 when initialize)
 | ret_c4:= START(pid1)
 | ret_c5:= START(pid2)
 | ret_c6:= START(pid3)
 | running:= when (active_partition_ID = Partition_ID)
 | diagnostic:= PROCESS_SCHEDULINGREQUEST(when running)
 | (active_process_ID,st):= PROCESS_GETACTIVE(running)
 | timedout:= UPDATE_COUNTERS()
 | timedout ^= running
 | ret_c7:=
         SUSPEND(active_process_ID when end_processing)
 |)
where
 ProcessAttributes_type att1, att2, att3;
 boolean running, diagnostic;
 ProcessStatus_type st;
 ReturnCode_type ret_c1, ret_c2... ret_c7;
 ProcessID_type pid1, pid2, pid3;
end;
```

This model is partially described using APEX services (process and time management), e.g. CREATE_PROCESS, START, SUSPEND. The other services used such as PROCESS_SCHEDULINGREQUEST, PROCESS_GETACTIVESTATUS, specify implementation dependent functions (e.g. scheduling policy). So, our library is not limited to APEX services only.

Now, let us look at the meaning of the example in a more detailed way. The presence of the input signal initialize corresponds to the initialization phase of the partition. Here, three ARINC processes identified by pid1, pid2, and pid3 are first created, and started just after (they correspond to the process POSITION_INDICATOR, FUEL_INDICATOR, and PARAMETER_REFRESHER in **Figure 5**).

The input active_partition_ID represents the identifier of the running partition selected by the module level OS[7], and it denotes an execution order when it identifies the current partition (this is expressed in the definition of the boolean running). Process rescheduling is performed whenever the partition is activated. This is done in the PROCESS_SCHEDULINGREQUEST service call.

The last input end_processing, is received from the contained processes (see **Figure 5**). It denotes the completion of a running process. On the occurrence of this signal, the running process is suspended (this is expressed by the SUSPEND call). The output active_process_ID returned by the PROCESS_GETACTIVE call, identifies the active process. It is designated by the OS with respect to the considered scheduling policy (priority preemptive). This signal is sent to all processes within the partition. Finally, time counters are updated using the UPDATE_COUNTERS service. The output timedout is sent to processes to notify them about the expiration of their associated time counters.

A SIGNAL program like the one depicted in **Figure 5** models an embedded application. Properties can be checked on this program using the POLYCHRONY tool-set. For instance, the clock calculus allows to solve synchronization constraints, thus proving the consistency of the specification, it also synthesizes a control structure of the application, which helps for instance for optimized embedded code generation. Other properties such as reachability or liveness can be addressed with the model checker SIGALI. Moreover, performance evaluation is enabled using program morphisms. One derives a temporal interpretation of the application (another SIGNAL program) which can be simulated on various platforms, in order to get timing information (like worst case execution times).

5. Discussion

The approach we have presented here embraces a component based philosophy. An immediate benefit is reusability. The modeled services could be easily adapted for applications based on other real-time standards (e.g. RealTime-Java). On the other hand, these models are not platform-specific. So, there is no risk of influencing non-functional properties of described applications.

As mentioned earlier, two major features of SIGNAL programming are modularity and abstraction. They play an important role for the scalability in our approach. Indeed, the description of a large application is achieved with respect to

[7]Similarly to the process model, an activation of each partition model depends on the input *active_partition_ID*, which identifies the current active partition. This signal is produced by the module level OS which is in charge of the management of partitions within a module.

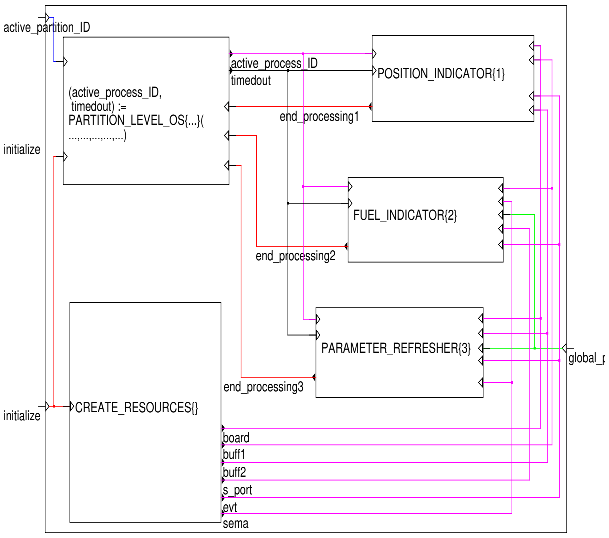

Figure 5. Example of partition model with POLYCHRONY.

a well-defined design methodology which consists in specifying either completely or partially (by considering abstractions) the sub-parts of the application. After this, the resulting components can be composed in such a way to derive new components. These components can be also composed in order to obtain other components and so on. Of course, at any stage during the descriptions, verification and analysis remain possible.

A crucial issue about the design of safety critical systems, like in avionics, is validation. Simulation is widely used in order to validate those systems. In POLYCHRONY, this is possible by generating a simulation code (e.g. C, Java). However, more sophisticated techniques and tools may be required to check other desired properties like safety. These can be addressed by the clock calculus (performed by the compiler) or model checking techniques (using SIGALI).

Finally, using SIGNAL as a single semantical model allows the description of an application at different stages of the design (from the specification of properties to the implementation). The transition between two representations at different stages is validated by transformations defined in the model. These transformations guarantee a certain traceability which can help the designer to analyze descriptions at any stage. This aspect will be useful for a reverse mapping issue. Some of these transformations are offered in the form of functionalities available within POLYCHRONY.

Related work. Among existing approaches which can be related to our work, we first mention TAXYS [3]. It is dedicated to the design and validation of real-time embedded applications. The specification of an application uses the languages ESTEREL and C to describe respectively the control and functional parts. The whole is instrumented with execution time (associated with the functional part) and deadline constraints. Then, it is compiled with an ESTEREL compiler to produce a model of the application analyzable with the model checker KRONOS for timing analysis. The similarity of TAXYS and our approach is that they both aim at taking advantage of the synchronous technology. On the other hand, in TAXYS, the use of timed automata allows schedulability analysis. However, while our approach is component-based, in TAXYS there are no pre-defined models for the description of applications.

Other approaches define specific languages for the design. This is the case for GIOTTO which is dedicated to embedded control systems [10]. It provides an abstract model of a system. Its compiler automates the implementation on a particular platform, and a runtime library which can be targeted toward various platforms. So, as in our case, GIOTTO specifications are not platform-dependent. The language has a time-triggered semantics. This facilitates time predictability for system analysis, but it is restricting since *tasks* are essentially periodic. The models we propose include both periodic and aperiodic processes.

The most popular Architecture Description Language for the design of real-time, distributed avionics applications is *MetaH* [18]. A user specifies how software and hardware pieces are combined to give the global system. The language tool-set generates formal models and executive, and performs analysis for schedulability, safety/reliabity, and correct partitioning. Both periodic and aperiodic processes are supported and the scheduling policy is preemptive fixed priority. In our case, the scheduling paradigm is the same. However, as mentioned earlier, the scheduling policy is implemented by a special service (which does not belong to original APEX services). A modification of this service does not affect neither the other services nor the process models. So, one can easily take into account another scheduling policy by modifying only the service. In MetaH, intertask communications occur at special points during computations: a sending task writes data into a port only after completion, and a receiving task reads data from a port only at the release time. While this exchange scheme avoids certain situations such as message loss due to overload, it could be restricting. The service models provided in our library allow a running process to read or write data at any instant, whenever the required communication mechanism is available.

The last approach we mention is PTOLEMY [13]. It is dedicated to the support of modeling, simulation, and design of concurrent systems. It particularly addresses embedded systems, and integrates a number of models of computation (e.g. synchronous/reactive systems, continuous time, etc.) which deal with concurrency and time. Like our approach, PTOLEMY adopts a component-based approach. In a certain way, our approach could be seen as a particular case of the PTOLEMY approach since SIGNAL adopts only a synchronous/reactive computation model. In PTOLEMY, the semantics of component interactions is dictated by the models of computation. So, the focus is on the choice of suitable models to get the needed behavior in the system. In our approach, emphasis is put on both behavioral and structural aspects in the system description. The system architecture components (e.g. OS, processes) are clearly identified. The whole is represented by a SIGNAL program on which performances can be evaluated (e.g. worst case execution times) by co-simulation or even formal analysis, using a technique of program morphism [11].

6. Conclusions

We have proposed a component-based approach to the modeling of avionics applications. The defined library mainly contains models of so-called APEX services defined by the avionics standard ARINC 653. Also, a model of executive units (ARINC processes) is proposed. A combination of the whole allows to model partitions. The synchronous language SIGNAL has been used for the specification. This allows to access the POLYCHRONY tool-set. It gives the possibility of high level specification, verification and analysis, automatic code generation, and strongly favors validation.

There are two ongoing applications of the approach presented in this paper. The first one concerns the modeling of a real-world avionics application in collaboration with Airbus (a partner in the SAFEAIR project). In the other application, we propose a translation of real-time Java programs into SIGNAL models. The aim of this work is also to take advantage of the POLYCHRONY tool-set for performance evaluation. It aims to show the suitability of our approach to describe applications based on other real-time standards.

Finally, another ongoing work concerns the definition of a way to associate timed models with our descriptions. This is common practice and often useful when coping with schedulability problems. For instance, the TAXYS approach uses timed automata for these problems whereas, hybrid automata are used in MetaH. In both cases, there are efficient tools to support analysis.

References

[1] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *proc. of the IEEE, vol. 79, No. 9, p. 1270-1282*, April 1991.

[2] A. Benveniste, P. Le Guernic, and C. Jacquemot. Programming with events and relations: the SIGNAL language and its semantics. In *Science of Computer Programming, 16:103-149*, 1991.

[3] 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 *proc. of Computer Aided Verification, LNCS 2102, Paris*, July 2001.

[4] A. E. E. Committee. Avionics application software standard interface. In *ARINC Specification 653, Annapolis, Maryland*, January 1997.

[5] A. E. E. Committee. Design guidance for integrated modular avionics. In *ARINC Report 651-1, Annapolis, Maryland*, November 1997.

[6] D. Engler, D. Andrews, and D. Lowenthal. Efficient support for fine-grain parallelism. In *tech. report, University of Arizona*, 1993.

[7] A. Gamatié and T. Gautier. Modeling of modular avionics architectures using the synchronous language SIGNAL. In *proc. of the WIP session, 14th Euromicro Conference on Real Time Systems, p. 25-28*. Vienna, June 2002.

[8] D. Goshen-Meskin, V. Gafni, and M. Winokur. SAFEAIR: An integrated development environment and methodology. In *proc. of INCOSE'01, Melbourne*, July 2001.

[9] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Publications, 1993.

[10] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Embedded control systems development with Giotto. In *proc. of LCTES. ACM SIGPLAN Notices*, 2001.

[11] 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, p. 6/1-6/9*. HP Labs, Bristol, February 1996.

[12] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. In *proc. of the IEEE, 79(9)*, pages 1321–1336, September 1991.

[13] E. A. Lee and al. Overview of the Ptolemy project. In *tech. 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), p. 325-346*, October 2000.

[15] A. Pnueli. Embedded Systems: Challenges in Specification and Verification. In *proc. of 2002 Conference on Embedded Software, J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS 2491, Springer Verlag, p. 252-265*, 2002.

[16] J. Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. In *tech. report, Computer Science Laboratory SRI International, Menlo Park CA 94025 USA*, March 1999.

[17] J. Sifakis. Modeling Real-Time Systems - Challenges and Work Directions. In *EMSOFT'01, Tahoe City. Lecture Notes in Computer Science 2211*, October 2001.

[18] S. Vestal. MetaH support for real-time multi-processor avionics. In *IEEE Workshop on Parallel and Distributed Real-Time Systems*, April 1997.