INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Synchronous Modeling of Modular Avionics Architectures using the SIGNAL Language*

Abdoulaye GAMATIÉ — Thierry GAUTIER

**N° 4678**

December 2002

THÈME 1


ISSN 0249-6399 ISRN INRIA/RR--4689--FR+ENG

# Synchronous Modeling of Modular Avionics Architectures using the SIGNAL Language

Abdoulaye GAMATIÉ, Thierry GAUTIER

Thème 1 — Réseaux et systèmes
Projet Espresso

Rapport de recherche n° 4678 — December 2002 — 127 pages

**Abstract:** This document presents a study on the modeling of architecture components for avionics applications. We consider the avionics standard ARINC 653 specifications as basis, as well as the synchronous language Signal to describe the modeling. A library of APEX object models (partition, process, communication and synchronization services, etc.) has been implemented. This should allow to describe distributed real-time applications using Polychrony, so as to access formal tools and techniques for architecture evaluation.

**Key-words:** Signal, ARINC, avionics architectures, real-time system modeling.

This work has been supported by the european project IST Safeair (Advanced Design Tools for Aircraft Systems and Airborne Software) [GMGW01] (`http://www.safeair.org/`).

Unité de recherche INRIA Rennes

# Modélisation Synchrone d'Architectures Avioniques Modulaires à l'aide du langage SIGNAL

**Résumé :** Ce document présente une étude sur la modélisation de composants d'architecture pour des applications avioniques. Nous considérons les spécifications de la norme avionique ARINC 653 comme base, ainsi que le langage synchrone SIGNAL pour décrire la modélisation. Une bibliothèque de modèles d'objets APEX (partition, processus, services de communication et synchronisation, etc.) a été mise en œuvre. Ceci devrait permettre de décrire des applications distribuées temps-réel en utilisant POLYCHRONY, de manière à accéder aux outils et techniques formels pour l'évaluation d'architectures.

**Mots-clés :** SIGNAL, ARINC, architectures avioniques, modélisation de système temps-réel.

# Contents

# Part I
# Modeling of Avionics Architectures using SIGNAL

## 1 Introduction

The design of complex systems usually involves multiple formalisms (among others, natural language, programming languages) and various tools (e.g. provers, simulators), each at a specific phase of conception (specification, coding or testing). The main drawback of such an approach is that the design and checking tasks are inherently long and hard. As a matter of fact, transitions between design phases are difficult and error prone. Furthermore, for any detected error at a given phase, one must be able to update the whole design chain.
In the particular case of distributed real-time systems, there are additional difficulties: on the one hand, such systems have to be separated efficiently into components, and communication mechanisms between these components must be provided; on the other hand, the validation of the whole is required.

Therefore, a key challenge in embedded real-time system design [Sif01] [Lee00] is to provide practitioners with reliable and affordable tools, and enabling technology to overcome the obstacles above.
Modeling is essential to the design activity. It allows more flexibility concerning decisions, it is unnecessary to have the actual system to make experiments. Among the advantages emphasized by [Sif01], we can mention the enhanced adaptability of models and their parameters; more general descriptions by using genericity, abstraction, behavioral non determinism; the possibility of applying formal methods for analysis and predictability.
Many formalisms for real-time systems modeling have been proposed. Among them, we mention timed models (e.g. *timed* extension of *Petri nets* [Sif77], *timed transition systems* [HMP91] or *timed automata* [AD94]), Architecture Description Languages (ADLs) [Cle96] (e.g. MetaH [Ves97]), or the Unified Modeling Language (UML) modeling concepts as in [SR98]. Related tools like KRONOS [Yov97] for timed automata enable verification of properties on models.

Over these past years, the synchronous technology [Hal93] 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. This includes possibilities of high level specifications, using *synchronous languages* [BB91]; modular verification of properties on these specifications; automatic code generation through formal transformations, and validation of the generated code against specifications. As a result, earlier architectural choices and behavioral simulation are enabled, and design ambiguities and errors can be significantly reduced.
The programming environment of the synchronous language SIGNAL [LGLL91], called POLY-

CHRONY, which is developed by INRIA[1] (`http://www.irisa.fr/espresso`), incorporates all these functionalities.

This report presents an approach to the modeling of real-time architectures in general, and modular avionics architectures in particular. We use the SIGNAL language to specify models. A previous paper [GG02] already introduced this work. Here, we give in a more detailed way, an improved version of the approach.
The remainder of the paper is organized as follows: Section 2 discusses design approaches for architectures in avionics. A short introduction to ARINC (Aeronautical Radio, Inc.) specification 653 [Com97b] is also given. Section 3 presents the main features of the synchronous language SIGNAL, while Section 4 focuses on our approach to modeling ARINC specifications in SIGNAL. Then, in section 5 we relate our approach to the literature. The conclusion and perspectives are given in section 6. Finally, an annex provides the specification of the modeled services.

## 2 Avionics architectures

Traditionally, avionics functions are implemented in such a way that each function has its dedicated fault-tolerant computer system. This architecture where systems are loosely coupled from each other is called *federated*. FIG. 1 gives a view of such an architecture. A great advantage is that fault containment is inherent to this approach. Unfortunately, there is a potential risk of massive use of resources: each function requires its own computer system, which is most of the time replicated for fault tolerance. Of course, overall costs are easily affected. For instance, installation and maintenance tasks are harder, weight and space become critical on board, etc.


Figure 1: Federated architecture: each function has its dedicated fault-tolerant computer system.

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

New architectural concepts have emerged in order to solve this problem, *Integrated Modular Avionics* (IMA) [Com97a] is one of them. In this philosophy, several avionics functions can be hosted on a single, shared computer system as shown on FIG. 2. A critical aspect for IMA is ensuring that shared computer resources are safely allocated so that there exists no fault propagation from one hosted function to another. This is addressed by the *partitioning* mechanism. It consists in a functional decomposition of the avionics applications, with respect to available time and memory resources. A *partition* [Com97b] is an allocation unit resulting from this decomposition. Partitioning promotes verification, validation and certification.

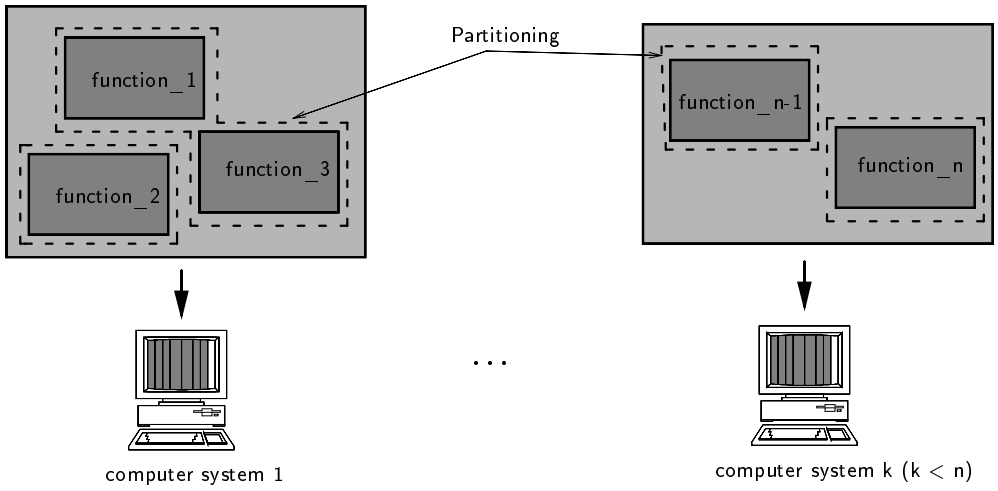



Figure 2: IMA architecture: several functions can be hosted on a single, shared computer system.

The *core module* encompasses partitions possibly belonging 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. This ensures a safe execution of applications. 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 core module or in different core modules. For instance, in FIG. 3, `Partition_2` and `Partition_3` are allocated to `processor_2`, `Partition_1` runs alone on the other processor. Partitions communicate asynchronously via logical *ports* and *channels*. There are two transfer modes in which channels may be configured: *sampling* mode and *queuing* mode. In the former, no message queuing is allowed. A message remains in the 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

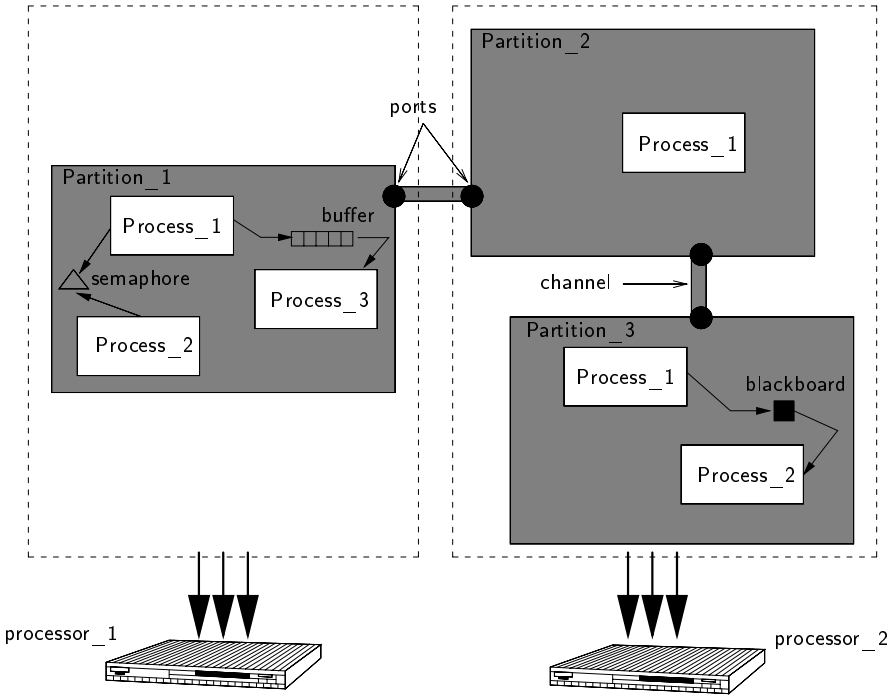



Figure 3: Example of three partitions running on 2-processors architecture.

destination port[2], until it is also overwritten. In the queuing mode, ports are allowed to store messages from a source partition in queues[3], until they are received by the destination partition. The message queuing discipline is First-In First-Out (FIFO). A set of services is provided to the application software for message exchange between partitions. In the sequel, these are referred to as *inter-partition communication services.*

Partitions are composed of *processes* which represent the elementary execution entities[4] (see FIG. 3). Processes run concurrently to achieve functions associated with the partition they are contained in. Each process is uniquely characterized by a set of information such as period, priority or deadline time, used by a partition level OS. Processes are scheduled through a priority preemptive policy: the process in "ready" state with the current highest priority executes whenever the partition is active. 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*, which provides controlled access to partition resources. Each mechanism is accessed via its associated service, e.g. *SEND_BUFFER, SET_EVENT, DISPLAY_BLACKBOARD, WAIT_SEMAPHORE.* In the sequel, we call them *intra-partition communication and synchronization services.*

The ARINC specification 653 [Com97b] relies on IMA. It defines the interface between the application software and the core software (OS, system specific functions), called APEX (APplication EXecutive). It is depicted in FIG. 4.

# 3 The synchronous language SIGNAL

The underlying theory of the synchronous approach [Hal93] 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 [BB91] are: ESTEREL [BG92], LUSTRE [HCRP91], SIGNAL [GLM94] [LGLL91]. They mainly differ from each other in their programming style. The first one adopts an imperative style whereas the two others are data-flow oriented (LUSTRE is functional and SIGNAL is relational). However, there had been joint efforts to provide a common format DC+ [BGMR97], which allows the interoperability of tools.

[2] A refresh period attribute applies to ports. When reading a sampling port, a *validity* parameter indicates whether the age of the read message is consistent with the required refresh period attribute of the port.

[3] A new instance of a message may carry uniquely different data. So, it should not be allowed to overwrite previous ones during the transfer.

[4] In fact, an analogy can be made between ARINC partitions and UNIX processes on the one hand, and ARINC processes and UNIX tasks on the other hand.

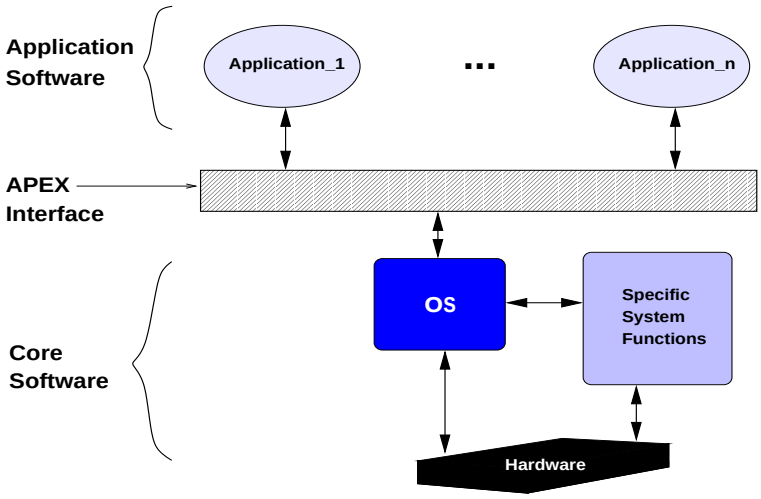



Figure 4: The APEX interface within the core module Software (from ARINC 653).

## 3.1 Main characteristics of SIGNAL

The SIGNAL language [GLM94] [LGLL91] handles unbound 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 kernel language.** SIGNAL relies on a handful of primitive constructs which are combined using a composition operator. These are:

- **Functions.** `y:= f(x1,...,xn)`, where $y_t \neq\perp\Leftrightarrow x1_t \neq\perp\Leftrightarrow \ldots \Leftrightarrow xn_t \neq\perp$, 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/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`.

**Extension.** The above core constructs are of sufficient expressive power to derive other constructs for comfort and structuring. We can mention the following ones:

- **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).
- **Memorizing**: we note two useful operators,
  - `y:= x cell b init y0`, defined as:
    `(| y:= x default (y$1 init y0) | y ^= ^x default (when b) |)`
    (`y` is equal to `x` when `x` is present. Otherwise, when `b` is `true`, `y` takes the latest value carried by `x`. One can notice that the clock of `y` depends on the ones of `x` and `b`)
  - `y := var x init y0`,
    Here also `y` always carries the latest value of `x`. The main difference from the above operator is that the clock of `y` is the one defined by the context in which it is used.

Another important characteristic of the SIGNAL language is the possibility of importing external objects (e.g. C++ functions). Thus, put together, all these features favor modularity and reusability. A complete and detailed description of all SIGNAL features will be found in [BGG02]. In the next sections, the use of the above operators is illustrated.

About the verification of SIGNAL programs, we can distinguish two kinds of properties: *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 the first one. 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* (see [LGLL91] for more details). Dynamical properties are addressed using other connected tools like SIGALI [MBLL00], a model checker.

## 3.2 Modularity: an important feature of the SIGNAL programming

Modularity is one major feature of SIGNAL-based specifications. Any process can be abstracted by an interface which specifies properties on its input-output signals. These properties essentially concern clock relations and dependencies between signals. Two kinds of abstraction are distinguished for a SIGNAL process: *black box* and *grey box* abstractions. In

the former, only properties on input-output signals are specified: it is shown on the left hand box of FIG. 5, the only visible information is how the outputs of a process P are related to the inputs.

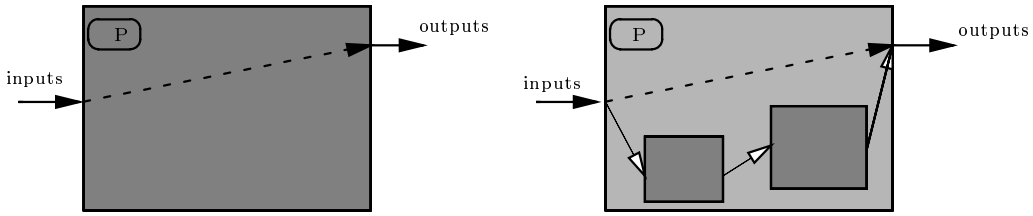



Figure 5: Black box (left) and grey box (right) abstractions of a process P.

The latter, represented by the right hand box, is a refinement of the former. In addition to the informations on the input-output signals of the process, it describes the possible internal interactions between sub-components which are black box abstractions in which inputs precede outputs (such that the outputs can be computed as soon as the inputs are available). The SIGNAL language provides a process frame which enables the definition of sub-processes. Sub-processes which are only specified by an interface without internal behavior (like the sub-components in the grey box abstraction) are considered as external (separately compiled processes or physical components). This is an essential feature in our approach.

In the sequel, we present how partitions are modeled in POLYCHRONY, and we also give an example for illustration.

# 4 Component modeling

To model a partition, we consider three basic elements:

1. the executive units which are ARINC processes;
2. the interactions (communication and synchronization) between these processes, which are achieved via the so-called APEX services;
3. the partition level OS, which is in charge of the correct concurrent execution of processes within the partition.

The executive model of a partition is mainly obtained by a combination of these elements as depicted in FIG. 6. In this section, we focus on the modeling of each of them: in sub-section 4.1, we show how APEX services are described in SIGNAL; then a model of ARINC processes is proposed in sub-section 4.2; and finally, sub-section 4.3 discusses the modeling of the partition level OS.

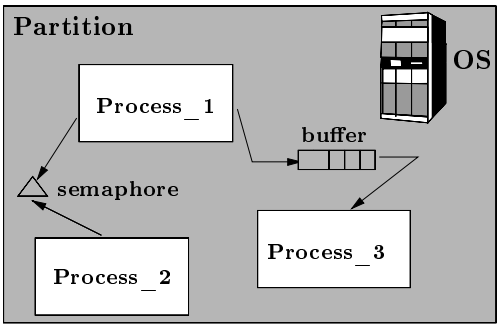



Figure 6: SIGNAL executive model of a partition.

## 4.1 APEX services

Here, we consider the following APEX services:

- **Communication and synchronization services** which describe how ARINC processes access mechanisms like buffers, events, blackboards, ports and semaphores. E.g. *SEND_BUFFER*, *WAIT_EVENT*.
- **Process management services**. E.g. *START*, *RESUME*, *STOP*.
- **Time management services**. E.g. *TIMED_WAIT*, *PERIODIC_WAIT*.

The design approach will be illustrated by considering one APEX service. We show how the corresponding SIGNAL model is obtained from informal specifications like those encountered in [Com97b].

**Modeling of an APEX service.** Let us consider the so-called *READ_BLACKBOARD* service, defined in [Com97b]. This service is used to read a message in a blackboard. The input parameters are the blackboard *identifier*, and a *time-out* duration for waiting whenever the blackboard is empty. The outputs are a *message* (defined by its address[5] 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 error handler* ***then***

[5] Also referred to as *area*.

```
    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 a "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 last available message of the blackboard and return NO_ERROR;
    end if
  end if
```

We have to derive a synchronous model which corresponds to the above informal specification. To see how this can be done, let us consider a concurrent execution of two processes P1 and P2 within a partition. 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 FIG. 7.
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 to switch and make P2 active. The process P1 must wait for either a notification that an initiated time counter becomes zero[6] (referred to as *situation A* on the figure), or the availability of some message (displayed by P2) in the blackboard (*situation B*). Now, if we check the timeline in both situations, we see that the time-lag corresponding to the READ_BLACKBOARD service is [`t2, t3`]. It partially includes both executions of P1 and P2. We remind that within a partition, only one process executes at any instant. In a synchronous view, it means that only statements associated with one process at most are executed within any step. Clearly, we have to split the specification of 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 when P1 is running (e.g. checking the validity of input parameters, initiating a time counter...), we call them *local actions*; and on the other hand, 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 are under the control of the so-called *partition level OS*, which is responsible for the management of all the processes, and common resources and mechanisms (blackboard, semaphore, time counters...) within the partition. That means each process blocked on a service request with a time-out will receive a *TIMED_OUT* signal from the

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

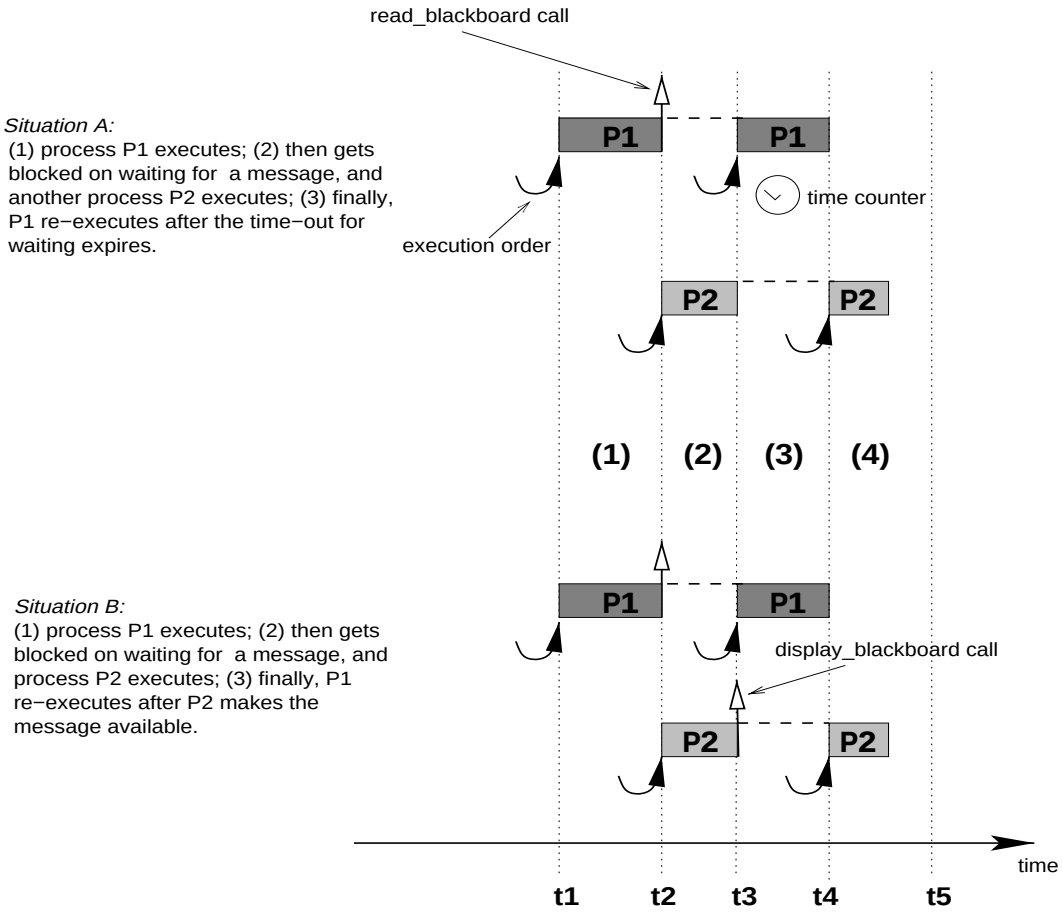



Figure 7: Concurrent execution of two processes P1 and P2 on one processor.

partition level OS model. The subsection 4.3 addresses the modeling of the partition level OS.

So, the modeling of a service consists in a **temporal split of its associated actions into subsets** in such a way that one can select which subsets have to be executed within a given synchronous step. This is a basic principle for the modeling of services.

Now, we show how the local actions of the READ_BLACKBOARD service are modeled using SIGNAL. For that, we consider the *situation B* of FIG. 7 where P1 resumes after P2 has displayed a message on the blackboard. Local actions (executed by P1) take place exactly at `t2` (e.g. check the validity of input parameters, initiate a time counter for waiting...) and `t3` (e.g. retrieve the last 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 READ_BLACKBOARD 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 FIG. 8.
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)` on the figure). The boolean `blocked` is set to *true* as soon as the resource is not available (empty blackboard). This 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 was previously blocked), the subset `L'` is executed and the boolean `blocked` becomes *false*.

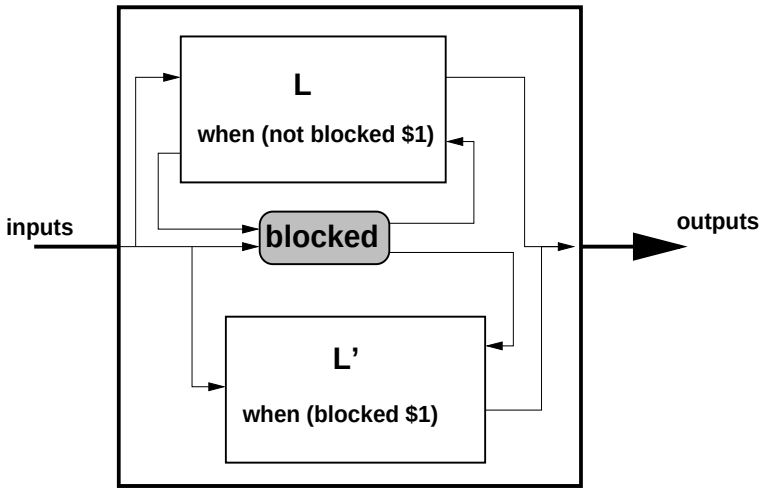



Figure 8: Rough model of local actions associated with a blocking service.

**Remark 1** *The model on* FIG. *8 is a general view, it can be simplified for some services. This is the case for the READ_BLACKBOARD service where* `L` *and* `L'` *denote the same actions. As a matter of fact, the calling process has to read again the blackboard in order to get the last displayed message. This is different from what is done when calling the RECEIVE_BUFFER service (see the annex B). The message retrieved by the resumed process is not necessarily the last available one in the buffer. It is the message whose arrival induced the release of the process. So, in such a case, the actions achieved in* `L'` *in order to get the message are not the same as those associated with* `L`.

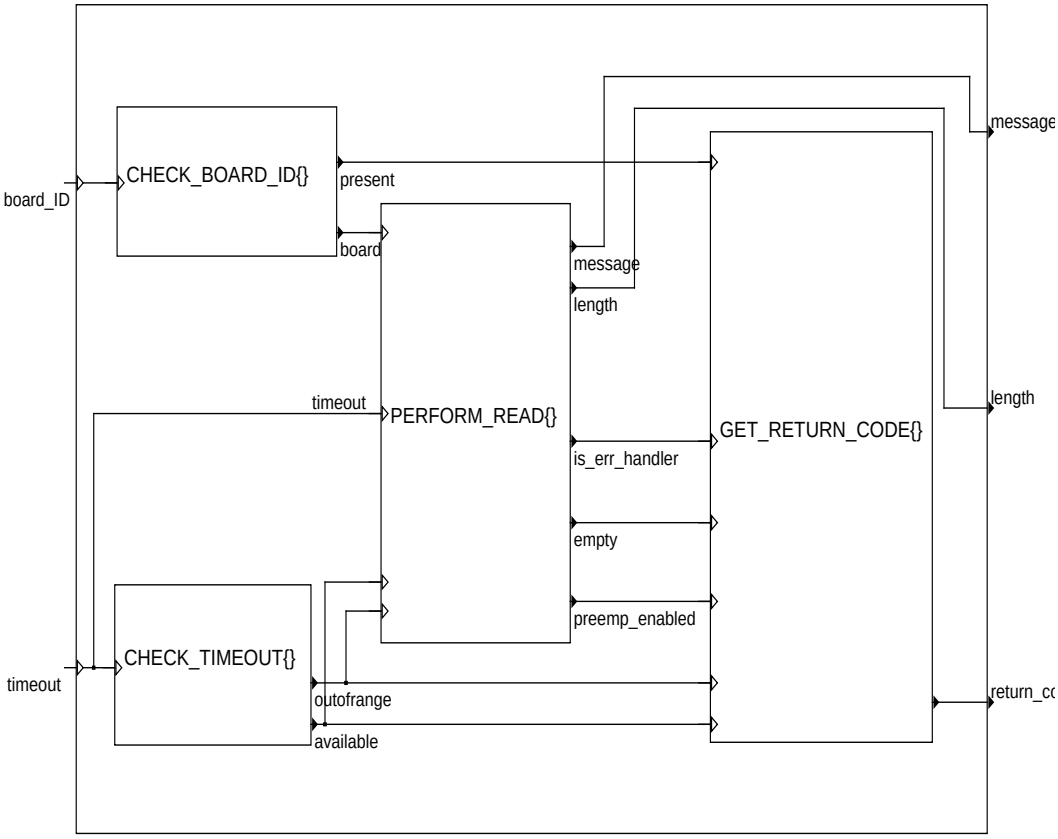
In the following, we mainly concentrate on the specification of a subset of actions (like `L` or `L'`). We bring out the capabilities of SIGNAL to allow modular specifications.

The SIGNAL process shown in FIG. 9 models local actions executed on a READ_BLACKBOARD service[7] request (i.e. `L` $\cup$ `L'`). There are four main sub-processes. This description follows the $2^{nd}$ *Design Principle*[8] as defined in the SIGNAL programming methodology [GLM94].
The sub-processes `CHECK_BOARD_ID` and `CHECK_TIMEOUT` verify the validity of input parameters `board_ID` and `timeout`. If they 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 respectively specified by `message` and `length`), and transmits all the necessary information 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 must wait for a message in the blackboard, and no other process can execute during the wait because the current operating mode does not allow preemption). 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 sub-processes are often defined by following the same design principle (e.g. `PERFORM_READ`).

In the abstraction, the keyword `spec` introduces the specification of properties on the input and output signals. For instance, the property `(s.2)` means that the return code is present only when a local boolean `C_return_code` is *true* (This boolean carries the value *false* when a time counter is initiated on a read request to wait for the availability of a message in the blackboard, and no return code occurs yet. Otherwise, it holds the value *true*. Since this boolean is defined internally, it simply appears in the abstraction as a local signal whose definition is not given: it may be interpreted as "there exists `C_return_code` such that the properties expressed in the abstraction hold"). In `(s.1)`, it is stated that this local signal is synchronous with the inputs of the service (i.e. whenever there is a read request, `C_return_code` indicates whether or not a return code must be sent). The lines `(d.1)` and `(d.2)` specify dependencies between the inputs and outputs. For example, the

[7] The complete SIGNAL code of this model is given in the annex A. Note that we call it READ_BLACKBOARD even though it only describes the local actions.

[8] "Decompose a process into functionally-coherent components".




```
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;
   )
  spec (| (| {{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 Cond_return_code;
;
```

Figure 9: SIGNAL model of the READ_BLACKBOARD service, and its corresponding abstraction.

property (d.2) states that the signals timeout and board_ID precede the output signals message and length when the diagnostic of the request is NO_ERROR.
The modeling of the other APEX services follows the same approach. We will see later that the decomposition of the informal specifications into local and global actions suits the computation model adopted for the ARINC process model (see the subsection 4.2).

**Remark 2** *In the interface of the READ_BLACKBOARD service on* FIG. *9, the parameter* process_ID *has been added. It does not exist in the original specifications [Com97b]. Here, it identifies the requesting process, and it is used when the process must be blocked and put in the waiting queue associated with the asked resource. The modeling of the other services with similar actions follows the same scheme.*

**Data structures for component management.** The management of components (e.g. communication mechanisms, ARINC processes...) is done via global data structures in which component attributes are collected. We call them *component managers*. They are consulted to verify whether or not a component specified in a user application actually exists; and modified in order to update the status of a component (e.g. the time capacity or the deadline time of an ARINC process may be changed by a user application request). The implementation of such data structures on which we often make side effects is slightly complicated in SIGNAL. We rather abstract them because side effect mechanisms do not exist at all in the SIGNAL language. So, they are first implemented as C++ classes (e.g. class BlackboardManager for the manager of blackboards). Then, the SIGNAL models associated with the methods of these classes can be defined. For instance, the service CHECK_BOARD_ID models the method that checks the presence of a blackboard identifier in the associated manager. To define it, we use *pragmas* which allow to associate specific information with the objects of a SIGNAL program (inputs, outputs and parameters). The information may be used further by the compiler or another tool. The example below illustrates the use of pragmas in the specification of the service CHECK_BOARD_ID.

**Example 1** *The* SIGNAL *process on* FIG. *10 describes the service that checks whether or not the blackboard identified by the input* board_ID *has been recorded in the associated manager. Such a service definition is also called external process. The output* present *is TRUE if the blackboard is present in the manager, then the other output* board_OUT *is the identified blackboard. Otherwise,* present *is FALSE.*
*In the spec section, the constraint* (s.1) *states that whenever the process is activated (that means the signal* board_ID *is received), there is an answer (the signal* present *is sent); the property* (s.2) *says that the signal* board_OUT *is present only when* present *is TRUE. Input/output dependencies are expressed by statements* (d.1) *and* (d.2).
*Pragmas are introduced by the keyword* pragmas. *Here, they are used for the purpose of code generation. The name* CPP_CODE *indicates that this pragma is interpreted when the implementation language is C++. The notations* &i1, &o1 *and* &o2 *encode respectively the first input, the first and second outputs. Finally, the whole statement in inverted commas (invocation of the method* BlackboardCheckID *applied to the object* GLOBAL_BLACKBOARD_MANAGER*)*

*represents the piece of code to be considered in the generated code, after the substitution of encoded parameters.*

```
process CHECK_BOARD_ID =
     ( ? blackboardID_type board_ID;
       ! boolean present;
         APEX_Blackboard_type board_OUT;
     )
  spec (| (| board_ID -> present                                (d.1)
          | { board_ID -> board_OUT } when present              (d.2)
          |)
       | (| board_ID ^= present                                 (s.1)
          | board_OUT ^= when present                           (s.2)
          |)
       |)
  pragmas
    CPP_CODE "&o1 = GLOBAL_BLACKBOARD_MANAGER->BlackboardCheckID(&i1, &o2)"
  end pragmas
```

Figure 10: Model of service defined using an abstraction.

**Remark 3** *One must take care when using pragmas in a service description. As a matter of fact, the statements in such a service are seen as a black box by the* SIGNAL *compiler. It means that these statements will not be concerned by the verifications performed by the compiler. For instance, in the* `CHECK_BOARD_ID` *service, only the constraints on the input/output signals will be relevant to the compiler. All what is done in the external method* `BlackboardCheckID()` *is transparent. So, the user must be sure that this piece of code is trustworthy. As a result, this may become penalizing since one does not take the maximum advantage of facilities offered by the* SIGNAL *programming. We remind that a major objective of modeling services in* SIGNAL *is to access to the formal techniques and tools available in* POLYCHRONY *for verification and validation.*
*Therefore, we have restricted as much as possible the use of external processes in our approach. Only services that concern the management of the global data structures are defined as external.*

All the modeled services are described in the annex B. For each service, we give on the one hand, an informal specification which roughly explains how it works; and on the other hand, a corresponding formal specification in the form of a SIGNAL abstraction which expresses properties on the input-output signals. So, in the annex B, we first introduce the allowable return codes used in the service descriptions (subsection 8.1). Then, the subsection 8.2 provides the description of the services dedicated to the management of processes. Intra-partition communication and synchronization services are given in subsection

8.3. They mainly concern the management and requests of mechanisms (buffer, event, blackboard, semaphore). Similarly, the services associated with the inter-partition communication mechanisms (sampling and queuing ports) are presented in subsection 8.4. Finally, subsection 8.5 discusses time management services.

## 4.2 ARINC process

Processes represent the executive units for an application. They share common resources (e.g. communication mechanisms like blackboard...), and execute concurrently within a partition. We describe their main features in the annex B (subsection 8.2). Here, we first define the associated SIGNAL model. Then, we show how this model is used to describe a concurrent execution like the one shown on FIG. 7. The definition of such a model takes into account two essential aspects: on the one hand, the *execution flow control* (i.e. how the actions are computed), and the description of *computed actions* (i.e. what the actions consist of) on the other hand. This is in agreement with the $3^{rd}$ *Design Principle*[9] of [GLM94].

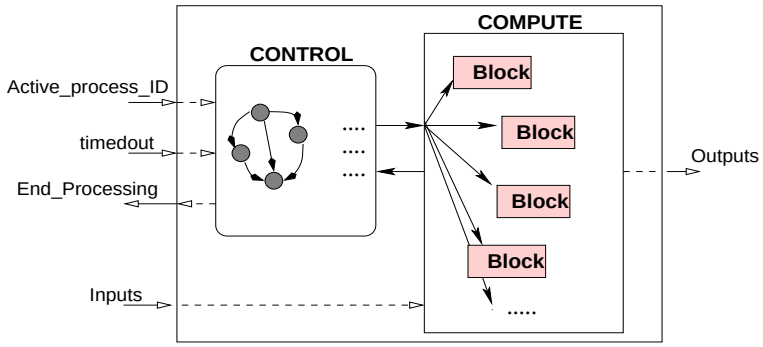



Figure 11: ARINC process model.

That is what we have depicted in FIG. 11. Two sub-components are clearly distinguished within the model: *CONTROL* and *COMPUTE*. This picture shows the ARINC process as a reactive component, where actions are performed whenever an execution order is received. The input signal `Active_process_ID` identifies the active process within the partition. It is sent to all the processes which must check it to see if they can execute. The signal `timedout` notifies the expiration of time counters to processes that may be blocked on a service request with time-out. Finally, the output `End_Processing` is emitted by a process after completion. In addition to these signals which always appear in the interface of the model, there can be other input (resp. output) signals needed for (resp. produced by) the process computations. The CONTROL and COMPUTE sub-components cooperate to achieve the right actions for every activation.

[9]"Split a process into computation and control parts".

**The CONTROL sub-component.** It specifies the execution flow of the ARINC process. Basically, it is a transition system that indicates which statements should be executed whenever the process is active. Automata are very easy to specify in SIGNAL.
A generic interface of the SIGNAL process that specifies the CONTROL sub-components is given on FIG. 12.

```
process CONTROL =
     { ProcessID_type PID... }
     ( ? ProcessID_type Active_process_ID;
         [NB_PROC]boolean timedout;
         info_type control_info_1 ... control_info_n;
       ! event end_processing;
         integer Active_block;
     )
     spec (| (| { { Active_process_ID, timedout,
                    control_info_1 ... control_info_n } ->
                    Active_block } when (Active_process_ID = PID)
              | { { Active_process_ID, timedout,
                    control_info_1 ... control_info_n } ->
                    end_processing } when C_end_processing
              |)
           | (| Active_process_ID ^= timedout ^= C_end_processing
              | control_info_1 ^< Active_process_ID
              | ...
              | control_info_n ^< Active_process_ID
              | end_processing ^= when C_end_processing
              | Active_block ^= when (Active_process_ID = PID)
              |)
           |)
     where boolean C_end_processing;
;
```

Figure 12: Generic interface of the CONTROL sub-component.

In this model, the parameter `PID` denotes the process identifier (the worst case execution time can be another parameter as well...). Thus, whenever `Active_process_ID` is the same as `PID`, the identified ARINC process executes. The input `timedout` is represented by an array of booleans such that `timedout`$[i]$ = TRUE when the time-out counter associated with the $i^{th}$ process comes to expire; otherwise, `timedout`$[i]$ = FALSE. The outputs `end_processing` and `Active_block` specify respectively the completion of the process and the *Block* of actions to be executed. The input signals `control_info_1` ... `control_info_n` are part of the

dialog between the CONTROL sub-component and the COMPUTE sub-component. They denote informations that are received from COMPUTE in order to take a "local control" decision (i.e. they represent labels in the automaton of the CONTROL sub-component).

**The COMPUTE sub-component.** It describes the actions computed by the process. It is composed of so-called *Blocks*. They represent elementary pieces of code to be executed without interruption like *Filaments* [EAL93], or *clusters*[10] [GL99]. All the statements within a Block are executed instantaneously in the synchronous model. This means that one must take care of what kinds of statements can be put together in a same Block. In our case, two sorts of statements can be distinguished.

- First, those which may cause an interruption of the running process. For instance, a communication service request like *READ_BLACKBOARD*, or a synchronization service request like *WAIT_SEMAPHORE* (used for accessing shared resources exclusively with a semaphore). 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*. A function affects only the local control of the process. For instance, depending on the result of a function executed in a Block $B_i$ and sent to the CONTROL sub-component, a decision is taken to choose either a Block $B_j$ or $B_k$ as the next statement to be executed.

FIG. 13 depicts a generic interface of the SIGNAL process associated with the COMPUTE sub-component.
The input signal `Active_block` indicates the current Block to be executed. The signals `control_info_1` ... `control_info_n3` are sent to the CONTROL sub-component, and `INPUT_1` ... `INPUT_n1` (resp. `OUTPUT_1` ... `OUTPUT_n2`) are additional inputs (resp. outputs) required (resp. produced) by actions specified in the Blocks.

**Execution of processes.** A process executes whenever it gets designated by the partition level OS. For that, CONTROL checks that the input `Active_process_ID` is the same as `PID`. Then, depending on the current state of the transition system representing the execution flow of the process, a Block is selected to be executed instantaneously. Afterwards, the process waits for a new execution order from the partition level OS, and so on. So, Blocks represent the execution units within processes.
We observe that this computation model is consistent with the decomposition for APEX services into local and global parts. Let us consider again the two processes of FIG. 7. The process model associated with P1 has a Block $B_r$ which contains the service request READ_BLACKBOARD. This Block is surrounded with other Blocks, which contain on the one hand the statements executed before the service request (in the portion `(1)` on

[10] Clusters are pieces of code associated with the black boxes representing the sub-components that result from a grey box abstraction of a SIGNAL process.

```
process COMPUTE =
     ( ? ProcessID_type Active_block;
         Input_type INPUT_1 ... INPUT_n1;
       ! Output_type OUTPUT_1 ... OUTPUT_n2;
         info_type control_info_1 ... control_info_n3;
     )
     spec (| (| { { Active_block, INPUT_1, ..., INPUT_n1 }
                     -> OUTPUT_1 } when C_OUTPUT_1
              | ...
              | { { Active_block, INPUT_1, ..., INPUT_n1 }
                     -> OUTPUT_n2 } when C_OUTPUT_n2
              | { { Active_block, INPUT_1, ..., INPUT_n1 }
                     -> control_info_1 } when C_control_info_1
              | ...
              | { { Active_block, INPUT_1, ..., INPUT_n1 }
                     -> control_info_n3 } when C_control_info_n3
              |)
           | (| Active_block ^= C_OUTPUT_1 ^= ... ^= C_OUTPUT_n2
                ^= C_control_info_1 ^= ... ^= C_control_info_n3
              | OUTPUT_1 ^= when C_OUTPUT_1
              | ...
              | OUTPUT_n2 ^= when C_OUTPUT_n2
              | control_info_1 ^= when C_control_info_1
              | ...
              | control_info_n3 ^= when C_control_info_n3
              |)
           |)
     where boolean C_OUTPUT_1, ..., C_OUTPUT_n2,
                   C_control_info_1, ..., C_control_info_n3;
;
```

Figure 13: Generic interface of the COMPUTE sub-component.

FIG. 7); and on the other hand, those executed after (i.e. in the portion (3)). Similarly, the model corresponding to P2 has a Block $B_d$ which contains the service request *DISPLAY_BLACKBOARD* (used to display messages on a blackboard).
On FIG. 14, we show an execution trace following our process model. It is another view of the scenario depicted in FIG. 7, for the *situation B*. The boxes represent Blocks. The

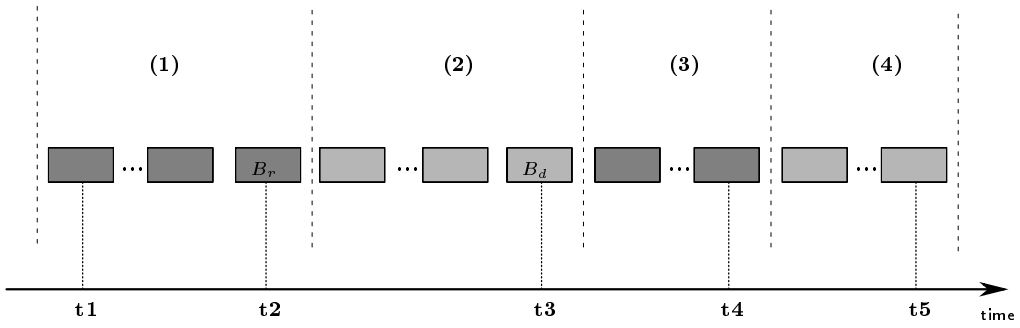



Figure 14: Trace resulting from a simulation of P1 and P2 models on one processor.

darkest ones belong to P1, and the others correspond to Blocks in P2. We have superimposed the execution paths for P1 and P2 to show how they interleave. We remind that each execution of a Block results from the receipt of an order of the partition level OS. We focus on the following sub-part of the trace: P1 executes first until t2; then, $B_r$ is computed, and the new active process becomes P2 until the computation of $B_d$. It illustrates the fact that the block $B_r$ represents only the local part of the READ_BLACKBOARD service. The global part is computed outside of $B_r$, during the execution of the process P2 in the portion (2). Finally, when P1 resumes, the other Blocks in its COMPUTE sub-component can be executed.

**Remark 4 (Block design precaution)** *From the above execution scheme, some design rules can be fixed about the statements that can be associated with a Block.*
***Rule 1:*** *Only one system call at most can be associated with a Block, and no other statement can follow this system call within the Block. The reason is obvious; a Block is executed instantaneously, so what happens if the system call interrupts the running process? All the other statements within the Block are executed in spite of the interruption, and this would not be correct. Moreover, when the process gets resumed, one does not need necessarily to re-execute the whole Block (e.g. only statements that follow the service call need to be executed). This will not be possible since any execution of the Block includes all the statements associated with this Block.*
***Rule 2:*** *The other rule concerns timing issues. The statements associated with a Block must complete within a bounded amount of time. This guarantees a real-time behavior.*

**Physical time.** To deal with the physical time, we make an important assumption: *the duration of a Block execution is one time quantum* (which represents the minimum amount of time during which a running process cannot be interrupted). It also corresponds to the number of physical time units that a simulation step takes. Thus, the time capacity allocated to an active process within a partition is decremented by one time quantum every simulation step. Also, the positive time-out counter values are decremented by this quantity of time.

**Preemption.** The process model we have proposed allows to interrupt an executing process and switch immediately to a higher-priority process whenever required. The active process can always change between two simulation steps (two Block executions). The choice of the active process depends entirely on the decision of the partition level OS which must support a priority preemptive scheduling policy.

As a general observation, the process execution model is *locally static* (the execution flow of an ARINC process is pre-determined, and described in its CONTROL component) on the one hand, and *globally dynamic* (the behavior within a partition depends on the evolution of the partition status, particularly the resource availability and processes status) on the other hand. Furthermore, the process model we have presented is "generic". It could be used as a model for another kind of execution support like threads in real-time Java for instance.

## 4.3 Partition level OS

The phase 1 of APEX [Com97b] does not assume dynamic process creation. So, all the processes associated with a partition must be created at an initialization phase (this is also the case for resources like communication mechanisms, etc.). Thereon, the partition can execute. Among the advantages of such an assumption, one can notice the higher degree of determinism of the system (e.g. processes can be guaranteed to have enough required resources).
The creation of processes is done within the partition level OS model using the APEX service *CREATE_PROCESS* (see the annex B for the description). Any created process needs to be started in order to be active. So, at least one process in the partition is started shortly after creation. This is achieved by using another APEX process management service, *START*. In a same way, the other process management services can be used in the partition level OS model to stop, suspend or resume a process, etc.
Another task of the partition level OS is to ensure the correct concurrent execution of processes within the partition (each process must have an exclusive control on the processor). We remind that the scheduling policy is priority preemptive. So, the OS has to manage the execution environment in such a way that processes are dispatched and preempted based on their respective priority and current status. Whenever a rescheduling is performed, the

process with the highest priority in the ready state is designated as the active one within the partition. In our library, this is achieved via two special services[11]:

- The service `PROCESS_SCHEDULINGREQUEST` which has the following abstraction:

```
process PROCESS_SCHEDULINGREQUEST =
     ( ? event sched_req;
       ! boolean diagnostic;
     )
     spec (| (| sched_req -> diagnostic |)
           | (| sched_req ^= diagnostic |)
           |)
;
```

  On receiving the input `sched_req` (an event resulting from either a direct request from an ARINC process or any partition internal event), a rescheduling is tried to be performed. The output `diagnostic` is set to `true` when the rescheduling has been actually performed. Otherwise, it is `false` (this happens when the partition prevents the process rescheduling operations of the OS).
  We can observe that only this service requires some modifications whenever a new scheduling policy has to be taken into account (e.g. Earliest Deadline First, Rate Monotonic Algorithm...). The original APEX services models are not affected at all, and their genericity is preserved.

- The other service `PROCESS_GETACTIVESTATUS` is defined as follows:

```
process PROCESS_GETACTIVESTATUS =
     ( ! ProcessID_type process_ID;
         ProcessStatus_type status;
     )
     spec (| process_ID ^= status |)
;
```

  This service is invoked after each rescheduling request to get the current active process within the partition. Its outputs `process_ID` and `status` respectively denote the identifier and status of this process.

On the other hand, the partition level OS model is in charge of the management of the time counters associated with processes in the partition. Each initiated time counter must be updated during the execution of processes. It is decremented until its value becomes zero, then a decision is taken about the process that initiated it. This is part of global actions resulting from an APEX service request like the READ_BLACKBOARD one when the

[11] They are not part of original APEX services.

calling ARINC process is interrupted. This is done via the special service `UPDATE_COUNTERS` defined as follows:

```
process UPDATE_COUNTERS =
     ( ! [MAX_NUMBER_OF_PROCESSES]boolean timedout;
     )
;
```

Every positive time counter value is decremented. When it becomes zero, `timedout[i]` is set to `true`, where `i` identifies the associated process. Otherwise, it is set to `false`.
An example of a rough partition level OS model is shown in FIG. 15.

```
process PARTITION_LEVEL_OS =
   { PartitionID_type Partition_ID; }
   ( ? PartitionID_type Active_partition_ID;
       event initialize;
       event end_processing;
     ! ProcessID_type Active_process_ID;
       [n]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_s1 := START{}(pid1) | ... | ret_s3 := START{}(pid3)
     | partition_is_running := (Active_partition_ID = Partition_ID)
     | diagnostic := PROCESS_SCHEDULINGREQUEST{}(
         when partition_is_running)
     | (Active_process_ID,status) := PROCESS_GETACTIVESTATUS{}()
     | timedout := UPDATE_COUNTERS{}()
     | Active_process_ID ^= timedout ^= when partition_is_running
     | ...
     |)
   where
     ProcessAttributes_type att1, att2, att3;
     boolean partition_is_running, diagnostic;
     ProcessStatus_type status;
     ReturnCode_type ret_c1, ret_c2, ret_c3, ret_s1, ret_s2, ret_s3;
     ProcessID_type pid1, pid2, pid3;
     [3]boolean timedout;
     ...
end;
```

Figure 15: An example of partition level OS model.

FIG. 16 depicts the relationship between the OS and processes within a partition. In this SIGNAL model, 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 models called `ARINC_process1`, `ARINC_process2` and `ARINC_process3` in FIG. 16).

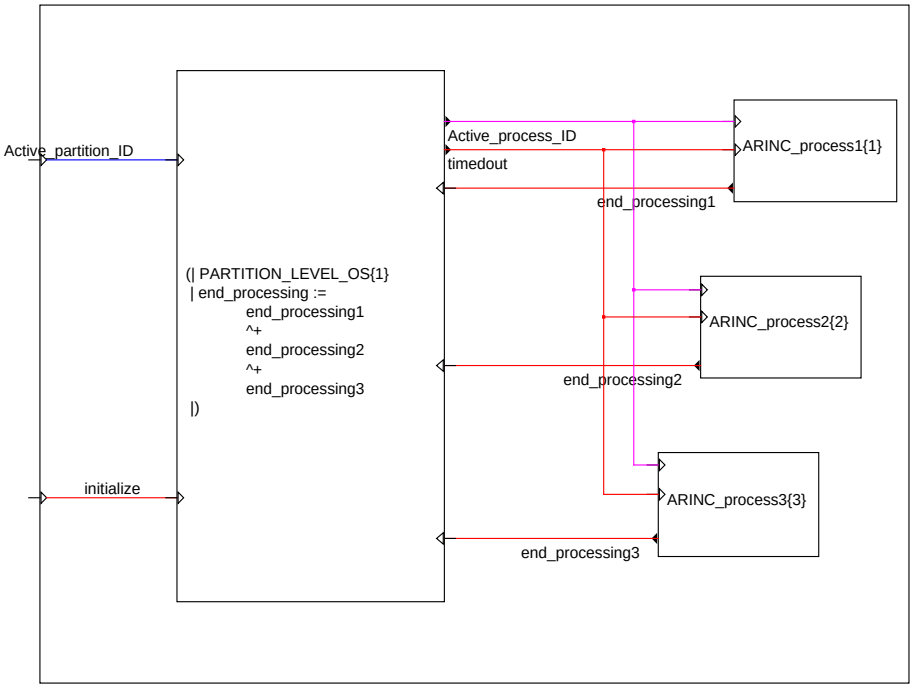



Figure 16: Interactions between the partition level OS and processes.

The input `Active_partition_ID` represents the identifier of the running partition selected by the module level OS[12], and it denotes an execution order when it identifies the current partition (this is expressed in the definition of the boolean `partition_is_running`). Process rescheduling is performed whenever the partition is activated. This is done in the `PROCESS_SCHEDULINGREQUEST` service call. As one can see in FIG. 16, the inputs `Active_partition_ID` and `initialize` are both external to the partition.

The input `end_processing` is received from any process of the partition which runs to completion. In FIG. 16, this signal is defined as the clock union of the signals

[12] As for the process model, an activation of each partition 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 the partitions in a module. The modeling of the module level OS is not addressed in this report.

`end_processing1`, `end_processing2` and `end_processing3`, received from the three ARINC processes of the partition. On an occurrence of this signal, a decision can be taken by the OS model about the next process to execute. Contrary to the other inputs of the partition level OS model, `end_processing` is an internal signal of the partition.

The output `Active_process_ID` identifies the active process whenever the partition is active. It is designated by the OS with respect to the considered scheduling policy. This signal is sent to all the processes within the partition.

Finally, time counters are updated through the `UPDATE_COUNTERS` service call whenever the partition executes. The output `timedout` is sent to processes to notify them the expiration of their associated time counters.

## 4.4 An example

To illustrate the use of the models, we consider a partition called `ON_FLIGHT`, depicted by FIG. 17. This small example takes its inspiration from a real world avionics application which is currently being modeled. The aim is to show how one can proceed to give the corresponding SIGNAL model using the approach presented before.
Roughly speaking, `ON_FLIGHT` is in charge of gathering informations concerning the current position and the fuel level of an aircraft during its flight. A report message is produced in the following format:

```
[date of the report : height : latitude : longitude : fuel level]
```

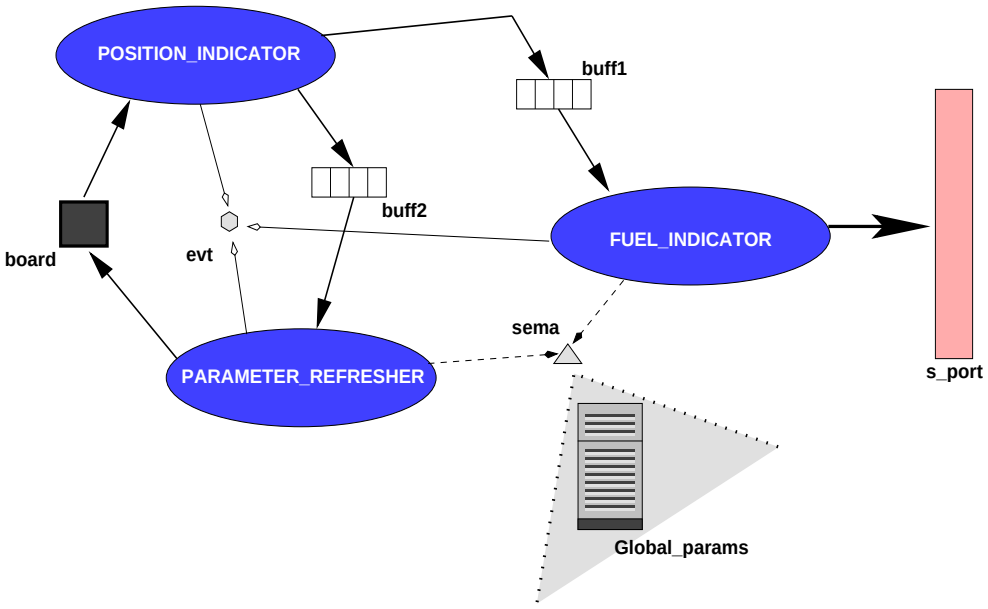



Figure 17: The partition ON_FLIGHT.

The partition is composed of the following objects.

- Communication and synchronization mechanisms: a blackboard `board`, two buffers `buff1` and `buff2`, an event `evt`, a semaphore `sema`, and a sampling port `s_port`.
- A resource `Global_params` which contains all the parameters needed by the associated avionics functions.
- Three processes defined below.

1. The process `POSITION_INDICATOR` produces a 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 global parameter refreshment, via buff2 (in order to be able to update the report message with position informations);*
   *wait for notification of end of refreshment, using evt;*
   *read the refreshed position values displayed on the blackboard;*
   *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 the process `FUEL_INDICATOR` is to update the report message with the current fuel level.

   ***if** a message is contained in the buffer buff1 **then***
   *retrieve this message;*
   ***end if***
   *update the report message 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;*
   *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;*

Since the modeling approach is the same for all three processes, we only give the detailed model of the process `POSITION_INDICATOR`, depicted by FIG. 18.

On FIG. 19, we have represented the *COMPUTE* sub-component of `POSITION_INDICATOR`. There are six Blocks. Each specifies a "one quantum" action. For instance, the second Block (from top to bottom) denotes a request for parameter refreshment, via `buff2`; and the third expresses the wait for a notification of the end of refreshment using *evt*.

**Remark 5** *We mention that there is no obligation to have only one level of diagram for a COMPUTE component. Doing this could lead to crowded boxes on the one hand, and space limitation on the other hand. The* SIGNAL *graphical editor allows to define a hierarchy of Block containers, which solves this problem.*

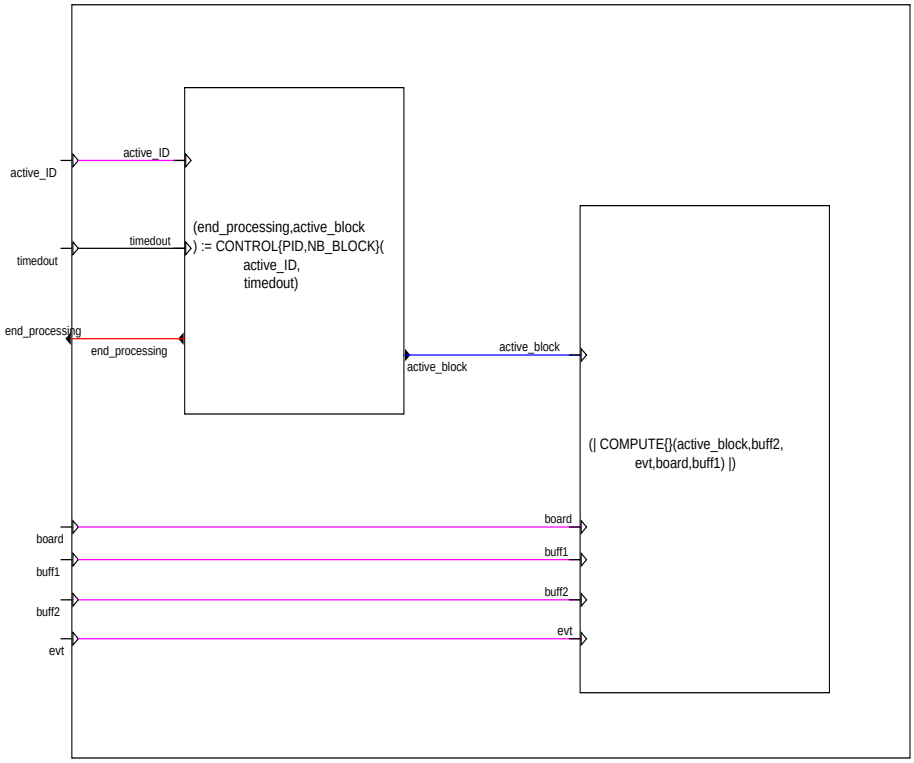



Figure 18: SIGNAL model of the process POSITION_INDICATOR.

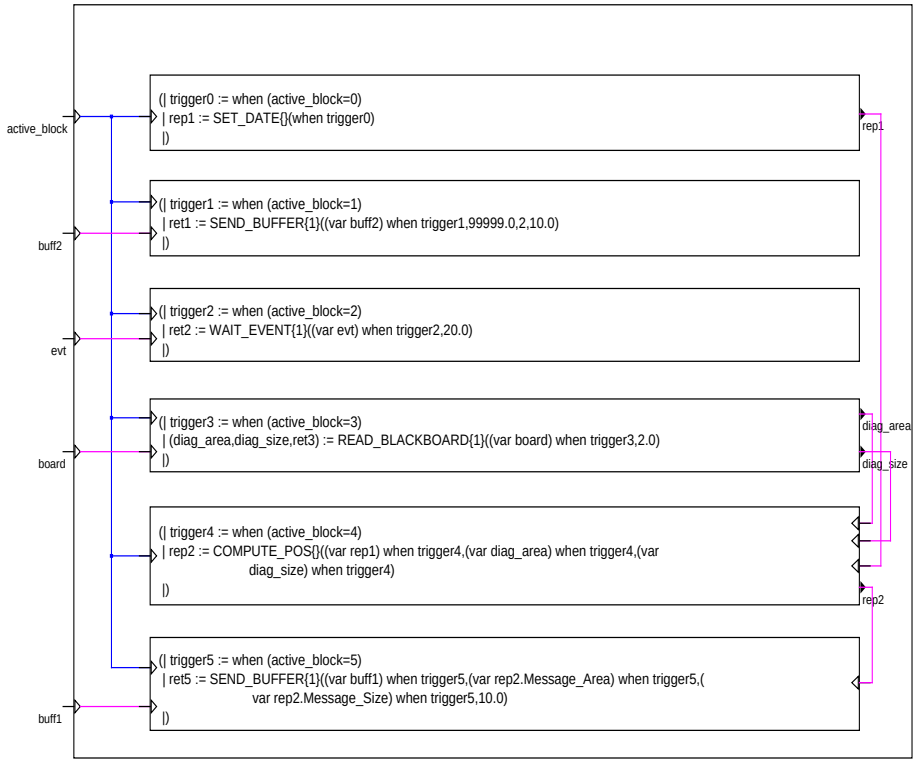
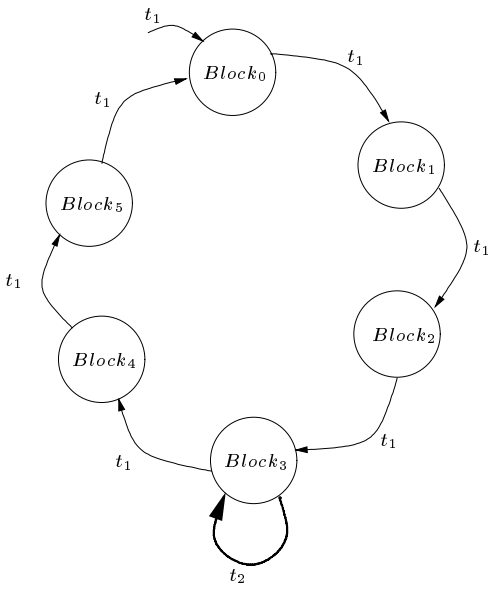



Figure 19: COMPUTE sub-component of the process POSITION_INDICATOR.

The *CONTROL* sub-component consists of the automaton depicted in FIG. 20. Blocks are executed sequentially (from top to bottom as shown in FIG. 19) whenever the process is active. We notice that the fourth Block, $Block_3$ (which contains a READ_BLACKBOARD service call) could be executed more than once successively (denoted by the transition $t_2$ on FIG. 20). This situation occurs when the process must retrieve a message from the blackboard after being blocked on waiting for this message.




$t_1$: this transition is fired whenever the process was not previously waiting for some message (which can be the case in in a read_blackboard service call).

$t_2$: this transition is fired whenever the process was previously waiting to retrieve some message.

Figure 20: Automaton associated with the CONTROL sub-component of the process POSITION_INDICATOR.

The corresponding SIGNAL model is given on FIG. 21. The signal `active_block` denotes the current state of the automaton. Its definition describes how transitions are fired. The end of the processing (denoted by the signal `end_processing`) is notified when the last Block is executed (state labelled $Block_5$ on the automaton).

Finally, FIG. 22 gives an approximate global view of the partition `ON_FLIGHT`. It includes the processes (`POSITION_INDICATOR`, `FUEL_INDICATOR`, and `PARAMETER_REFRESHER`); a subprocess `CREATE_RESOURCES`, which defines all communication and synchronization mechanisms used by processes; and the `PARTITION_LEVEL_OS` component. The whole model works as follows:


Figure 21: SIGNAL model of the CONTROL sub-component of the process POSITION_INDICATOR.

- When the input `initialize` is received, all the partition resources (processes, communication mechanisms, etc.) and the data structures are created. Then, the partition is set in the `NORMAL` operating mode[13]. This should be done at the beginning of any simulation.

[13]There are four operating modes [Com97b]: 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 input `active_partition` coming from the module level OS, specifies the active partitions. Within each running partition, the OS designates a process to execute. Finally, the *CONTROL* sub-component associated with this process selects the right Block to be computed. And so on.

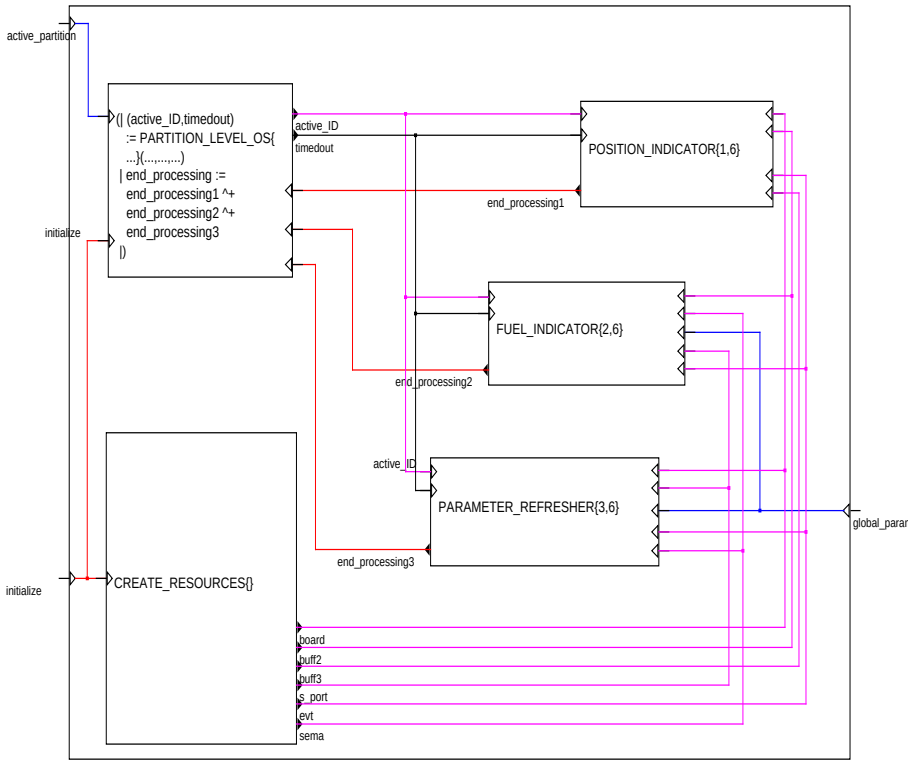



Figure 22: SIGNAL model of the partition ON_FLIGHT.

In FIG. 23, we show an execution trace of the partition ON_FLIGHT. The process POSITION_INDICATOR has the highest priority, and PARAMETER_REFRESHER has the lowest one.

the *NORMAL* mode, the scheduler is activated, and all the required resources in the partition must have been created before.

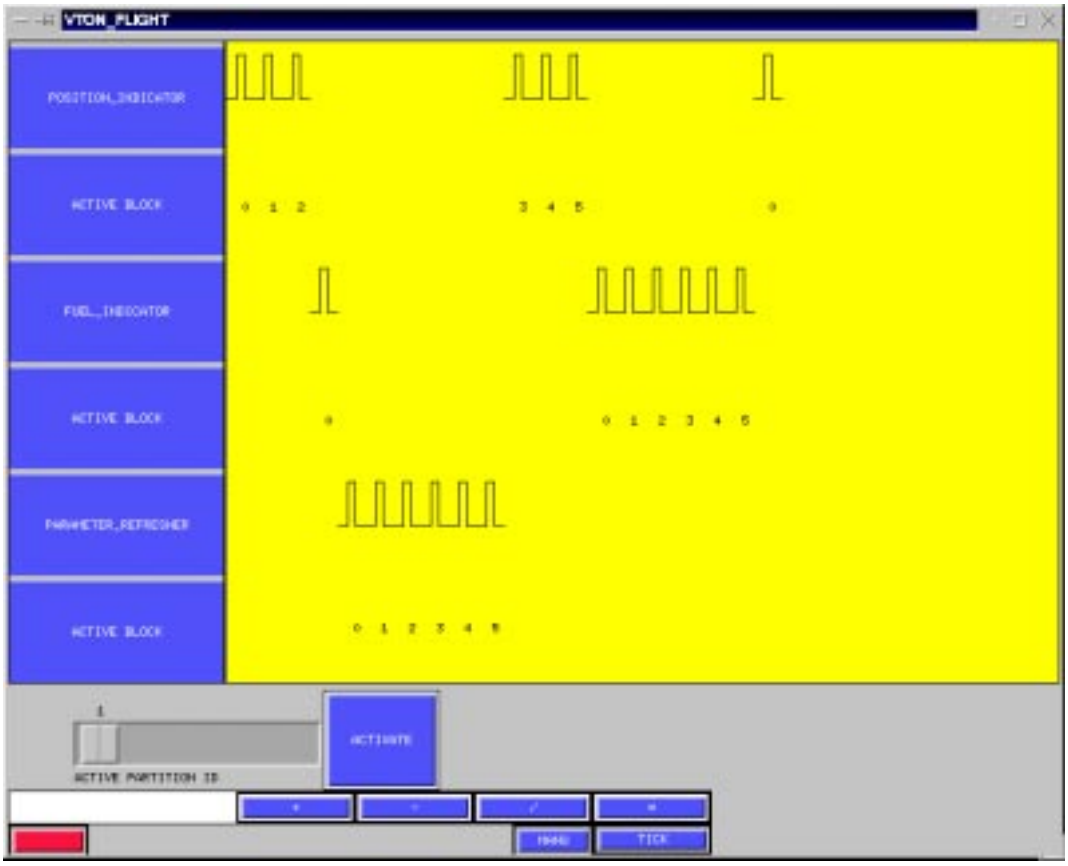



Figure 23: A simulation of the partition ON_FLIGHT.

# 5 Discussion

We have presented an approach to the modeling of avionics systems using the synchronous language SIGNAL. This approach embraces a component based philosophy. A major benefit of this kind of approach in system design is the reusability of models. For a particular system, architectural choices can be assessed earlier using existing models. This considerably decreases the overall development costs and time. Furthermore, the defined services could be easily adapted to the description of other applications based on other real-time standards (e.g. RealTimeJava).
Finally, the models are not platform-specific. So, there is not any risk of influencing non-functional properties of described applications. Components can be annotated with execution information (e.g. Worst Case Execution Times (WCET)) for a specific platform.

We can observe through the approach that the SIGNAL language acts as an Architecture Description Language (ADL) [Cle96]. The designer's perspective is shifted from small-grained features (such as lines of code) to large-grained features (such as process, partition...) with the suitable interaction mechanisms. Therefore, the question of SIGNAL as candidate to be an ADL may rise here.

One crucial question about safety critical systems, like in avionics, is how to be convinced of the correctness of the design. This is addressed by the validation process. The use of a formalism with a well-defined formal semantics for descriptions eases an answer to the question. This is another advantage of using the SIGNAL language. In essence, the family of synchronous languages which SIGNAL belongs to, relies on solid mathematical foundations. For other modeling languages such as UML, which suffer of the lack of a clear formal semantics, validation is merely difficult.
Simulation is widely used for validating real-time systems. In our case, simulation is possible by generating a target code (e.g. in C, Java...) from high level specifications through formal transformations. Then, the sequences of stimuli and responses of our reactive models can be checked to see whether or not the requirements are met.
However, simulation is not always sufficient to guarantee the correctness of designs. As a matter of fact, the set of design states actually explored is less than the set of all possible states. Therefore, simulation can miss some pertinent behavior.
Furthermore, some desired properties like safety cannot be checked using only simulation. We need more sophisticated techniques (e.g. abstractions) and tools (e.g. model checker) for that purpose. In POLYCHRONY, the compiler uses abstractions to check static properties (e.g. consistency of specifications), and tools like SIGALI (a model checker) helps for the verification of other kinds of properties (e.g. safety, liveness).

Finally, the use of a single semantical model (the SIGNAL's one) allows to describe a given application at different stages of the design (from the specification of properties to the implementation on a multi-processor architecture). The transition between two representations

from different stages is validated by transformations defined on the model. Some of these transformations are offered in the form of functionalities available within POLYCHRONY.

## 5.1 Related work

There are many approaches to modeling real-time embedded systems. Here, we discuss those which can be closely related to our work.

We first mention the TAXYS approach [CPP+01], defined for the design and validation of real-time embedded applications. The specification of such applications uses the synchronous language ESTEREL [BG92] and the C language to describe respectively the control and the functional parts. The whole is compiled with the ESTEREL compiler SAXO-RT [WBC+00] to produce a model of a given application analyzable by the model checker KRONOS [Yov97], for timing analysis.
Conceptually, the modeling of a system in TAXYS is seen as the composition of three automata describing: the application itself, the environment which is responsible of the activation of the application, and an external event-handler (which ensures correct interactions between the application and the environment). The ESTEREL program of the application is instrumented with execution time (associated with the functional part) and deadline constraints. These real-time annotations are used further by KRONOS for timing analysis for validation.
Similarly to our approach, TAXYS aims at taking advantage of facilities of the synchronous technology using the ESTEREL language. Moreover, the translation of the ESTEREL specifications into timed automata allows timing analysis. However, while our approach is component-based, in the TAXYS approach there are no pre-defined models for the description of applications.

Other approaches define specific language for the design. This is the case for the GIOTTO language [HHK01], which is dedicated to embedded control systems. It provides an abstract model of such systems; and its compiler automates the implementation of the systems (computation and communication schedules) on a particular platform, and a runtime library which can be targeted toward various platforms. So, as in our case, GIOTTO's specifications are not platform-dependent.
It has a time-triggered semantics. This facilitates the time predictability for system analysis, but the expressiveness of the language is limited 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* [Ves97]. It provides a set of tools for the design of real-time high assurance, distributed avionics software. A user specifies how software and hardware pieces are combined to give the global system. The language toolset generates formal models and executive, and performs analysis for schedulability, safety/reliabity, correct partitioning. Both periodic and aperiodic processes are supported and the scheduling policy

is preemptive fixed priority.
In the case of ARINC models presented here, the scheduling paradigm is the same. However, as we discussed earlier, it can be replaced by another one since process models are not bounded to the scheduling policy. Thus, unlike in *MetaH*, implementation constraints can be easily avoided.
In MetaH, inter-task communications occur at special points during computations: a sending task writes a data in a port only after completion, and a receiving task reads a data from a port only at the release time. This exchange scheme avoids certain situations like message loss due to overload for instance. But, it limits the richness of the description of possible inter-task interactions. Our models of APEX communication mechanisms (e.g. buffer, blackboard...) allow any active process to read or write data at any instant, as long as the resource is available. So, more inter-process interactions can be described.

The last approach we mention is PTOLEMY [La01]. It is dedicated to the support of modeling, simulation, and design of concurrent systems. It particularly addresses embedded systems [Lee00], and integrates a number of models of computation (e.g. synchronous/reactive systems, communicating sequential processes (CSP), continuous time, finite state machines) which deal with concurrency and time. Like our approach, PTOLEMY also adopts a component based design. Our approach could be seen as a particular case of the PTOLEMY's one since SIGNAL adopts only a synchronous/reactive computation model. However, the clock calculus allows the desynchronization of programs for a safe deployment on asynchronous architectures.
In PTOLEMY, component interaction semantics is dictated by models of computation. It mainly focuses on the choice of suitable models to get the needed behavior in the whole system. Our approach puts emphasis on both behavioral and structural aspects of a system model. The system architecture components (OS, processes, calls) are clearly identified.
Finally, while PTOLEMY proceeds by simulation to evaluate the performances of systems, POLYCHRONY includes another technique for performance evaluation in addition to simulation. This technique relies on a morphism of SIGNAL programs [KL96] which yields a temporal interpretation of a given program. Thus, it can be used to estimate the worst case execution time.

## 5.2 Tool support for real-time embedded systems modeling

Numerous tools exist for the modeling and analysis of real-time embedded systems. We can mention MODELBUILD[14] which has been developed in the SAFEAIR project [BDL+02]. It is built on the top of the industrial product SILDEX. MODELBUILD promotes component-based designs by providing wide libraries of basic components for the description of complex systems. Some of them are: the so-called GALS library which is used for GALS (Globally Asynchronous, Locally Synchronous) systems [BCL00]; the RTOS palette where some of the models are based on the concepts defined in [KRP+93]. MODELBUILD supports RMA (Rate

[14]It has been developed by TNI-Valiosys (`http://www.tni-valiosys.com`).

Monotonic Analysis) for timing analysis. The default formalism of MODELBUILD is SIGNAL, therefore the integration of our models is facilitated.

Another tool is TIMEWIZ [Corb], which is a product of the TimeSys company[15]. TIMEWIZ is dedicated to the modeling, analysis, and simulation of the execution of real-time systems. Models are based upon the notions of resources, events, and actions as described in [KRP+93]. They are built graphically using resource palettes, or by importing log files generated from visualization tools. Worst-case performance analysis using the techniques of rate monotonic analysis [KRP+93] can be achieved on the modeled system. Also, the average performance of the system can be understood by running discrete event simulations.
Finally, we mention RATIONAL ROSE RT [Cora] which is based on the industry standard UML, RHAPSODY from iLogix[16] which also uses UML, SCADE from Esterel Technologies[17], etc.

# 6 Conclusion

We have proposed a component-based approach to the modeling of real-time architectures for avionics systems. A library of components has been defined. It mainly contains models of so-called APEX (APplication EXecutive) services described by the avionics standard ARINC 653. The synchronous language SIGNAL has been used for the specification of the models. This allows to accede to the facilities offered by the language itself and its programming environment, POLYCHRONY. Among these, we mention high level specifications relying on solid formal semantics, tools for efficient code generation, property verification, and performance evaluation. Therefore, such a context must favor the validation activity which is a major goal of our study.
The library mainly includes components required for the description of a real-time execution platform: communication and synchronization mechanisms (e.g. buffer, port, semaphore), executive units (ARINC process), services used for the management of executive units (start, suspend, stop, resume...). Some of these component models do not especially depend on the ARINC standard. For instance, the model we have proposed for ARINC processes can be used to describe other kinds of executive units like real-time Java threads. So, their reusability in other contexts is facilitated. The library has still to be completed with models of the APEX services that we did not mention in this report (e.g. health monitoring services like the *RAISE_APPLICATION_ERROR* service, used to invoke an error handler process for a specific error code; partition management services).
There are two ongoing applications which should allow us to make the approach effective.
The first concerns the modeling of a real-world avionics application in collaboration with Airbus, which was one of our partners 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 access the formal techniques and tools available

[15] http://www.timesys.com
[16] http://www.ilogix.com
[17] http://www.esterel-technologies.com/v2

within POLYCHRONY. The translation uses the existing component models. It also shows the suitability of our approach for the description of architectures based on other real-time standards, particularly the real-time Java API. These are all opportunities to make significant experiments in order to evaluate the approach.

This approach contributes to improve the design methodology for distributed applications, previously proposed for SIGNAL in [GG99]. It consists of formal transformations of an initial program describing an application. These transformations preserve the original program semantics, and yield a final program that reflects the target architecture which is composed of a set of possibly heterogeneous execution components (processors, microcontrollers...). Further degrees of refinement of the description may be required for a better architecture-adaptation: for example, concerning communications or the type and nature of the links (that could be implemented using shared variables, synchronous or asynchronous communications...). If the target architecture features an OS, the needed model consists basically in the profile of the corresponding functions. For instance, according to the degree of use of the OS, we need models of synchronization gates, communications (possibly including routing between processors) or tasking functions (in the case of un-interruptible tasks: start and stop; in the case of interruptible tasks: suspend and resume, assignment and management of priority levels). In this context, the APEX objects (process, communication and synchronization mechanisms...) presented here can be used in the description.

The use of this approach also allows performance evaluation following the program morphism technique [KL96] implemented within POLYCHRONY. So, from the SIGNAL description of a partition (such as ON_FLIGHT in section 4.4) that models a real-time executive, we can derive another SIGNAL program which represents the corresponding temporal interpretation. Then, it can be simulated in order to estimate the worst case execution time of the whole application.

Finally, another ongoing work concerns the definition of a way to associate timed models with our descriptions. This is common practice and often useful coping with schedulability problems. For instance, the TAXYS approach uses timed automata for that; whereas in MetaH, hybrid automata are used. In both cases, there are efficient tools to achieve analysis. We have chosen timed automata for our models, so we can take advantage of the availability of powerful tool (KRONOS) and theories (e.g. scheduler synthesis methods [AGS02]).

# Part II
# Annexes

This annex is organized as follows: *annex A* presents the complete SIGNAL specification of the read_blackboard service. Then, in *annex B*, we give the informal specifications of the available services. Moreover, the corresponding formal specifications are provided in the form of SIGNAL abstractions. These services include the original APEX services that we have modeled in SIGNAL, they are denoted with a star symbol in exponent (e.g. READ_BLACKBOARD$^{\star}$). They also include the other services that we have added to complete the library. Finally, the implementation architecture is described in *annex C*.

## 7 Annex A: detailed SIGNAL program of the read_blackboard service

The following textual SIGNAL model corresponds to the APEX read_blackboard service that we have described in sub-section 4.1 to illustrate the modeling of APEX services.

```
process READ_BLACKBOARD =
     { ProcessID_type process_ID; }
     ( ? Comm_ComponentID_type blackboard_ID;
         SystemTime_type timeout;
       ! MessageArea_type message;
         MessageSize_type length;
         ReturnCode_type return_code;
     )
   (| (present,blackboard_in) := BLACKBOARD_CHECKID{}(blackboard_ID)
    | (| outofrange := timeout>MAX_TIMEOUT_VALUE
      | available := timeout>0.0
      |)
    | (| (| empty := blackboard_in.Empty_Indicator when (not outofrange)
         | err_handler := PROCESS_CHECKERRORHANDLER{}(process_ID when empty when available)
         | enabled := PROCESS_CHECKPREEMPTION{}(when (not err_handler))
         |)
      | (| PROCESS_SETSTATE{}(process_ID when enabled,#WAITING)
         | INSERT_BLACKBOARDQUEUE{}(process_ID when enabled,blackboard_in when enabled)
         | START_COUNTER{}(process_ID when enabled when (not (timeout=INFINITE_TIME_VALUE)),
                           timeout when enabled when (not (timeout=INFINITE_TIME_VALUE)))
         | diagnostic := PROCESS_SCHEDULINGREQUEST{}(when enabled)
         | mess := blackboard_in.Message when (not empty)
         |)
      |)
    | (| message := mess.Message_Area
      | length := mess.Message_Size
      | return_code := (#INVALID_PARAM when ((not present) or outofrange)) default
                       (#NOT_AVAILABLE when empty when (not available)) default
                       (#INVALID_MODE when ((not enabled) default err_handler)) default
                       #NO_ERROR when (not empty)
```

```
    | blackboard_ID ^= timeout
    |)
 |)
where
APEX_Blackboard_type blackboard_out, blackboard_in;
event blocked;
boolean diagnostic, present, outofrange, available, enabled, empty, err_handler;
Message_type mess;
end;
```

# 8 Annex B: Specification of the services

This section is organized as follows: sub-section 8.1 presents the possible return code values of a service request. Then, sub-section 8.2 describes the main features of an ARINC process (attributes, states). It also provides the services related to process management (usable in the partition level OS model). Intra-partition communication and synchronization services are given in sub-section 8.3. They concern on the one hand, the management of the mechanisms (buffer, event, blackboard, semaphore); and on the other hand, how these mechanisms are acceded by processes. In sub-section 8.4, we present the services associated with the inter-partition communication mechanisms (sampling and queuing ports). Finally, sub-section 8.5 discusses time management services.

## 8.1 Common types

There is a common type shared by all the modules (process management services, communication mechanism services...): the return code type denoted by *ReturnCode_type*. The allowable codes and their descriptions are given in the following table:

| **Value** | **Description** |
|---|---|
| NO_ERROR | valid request and performed operation |
| NO_ACTION | system's operational status unaffected by request |
| NOT_AVAILABLE | the request cannot be performed immediately |
| INVALID_PARAM | parameter specified in request invalid |
| INVALID_CONFIG | parameter specified in request incompatible with current configuration (as specified by the system integrator) |
| INVALID_MODE | request incompatible with current mode of operation |
| TIMED_OUT | time-out associated with request has expired |

## 8.2 Processes

For each process, there is a corresponding descriptor which contains all the information about its current status (attributes, current priority, current state, ...). The scheduler (implemented in the OS model) embodies a list of descriptors that are useful for the management of processes within a partition. In the sequel, we first precise the main features of ARINC processes. Then, the associated services are presented. They concern the process descriptor access and process management services.

### 8.2.1 Main features of a process

Each process is associated with two kinds of *attributes*: *fixed* and *variable* attributes.

1. Fixed attributes are:
   - Name: identifies uniquely each process within a partition.
   - Entry point: starting address of the process.
   - Stack size: overall size for runtime stack of the process.

- Base priority: capability of the process to manipulate other processes.
- Period: period of activation for a periodic process. A distinct and unique value should be specified to designate the process as aperiodic.
- Time capacity: elapsed time within which the process should complete its execution.
- Deadline: "hard" or "soft".

2. Variable attributes are:
   - Current priority: defines the priority with which the process may access and receive resources. It is set to basic priority initially and is dynamic at runtime.
   - Deadline time: indicates whether the process is satisfactorily completing its processing within the allocated time.
   - Process state: current scheduling state (dormant, ready, running, waiting) of the process.

Moreover, each process is characterized by a *state*, which indicates its current status. Four states are distinguished:

(a) Dormant: ineligible to receive processor resources. Before being started and after being terminated (or stopped).
(b) Ready: eligible for scheduling. When it is able to be executed.
(c) Running: currently executing on the processor. Only one process can be executing at any time.
(d) Waiting: not allowed to receive resources until a particular event occurs. This happens in the following situations:
   - waiting on a delay,
   - or waiting on a semaphore,
   - or waiting on a period,
   - or waiting on an event,
   - or waiting on a message,
   - and / or suspended (waiting for a resume).

The transitions between states as depicted in FIG. 24 are the following:

`Transition (1)`: the process is started by another process within the partition.
`Transition (2)`: the process is stopped by another process within the partition.
`Transition (3)`: the process is selected for execution.

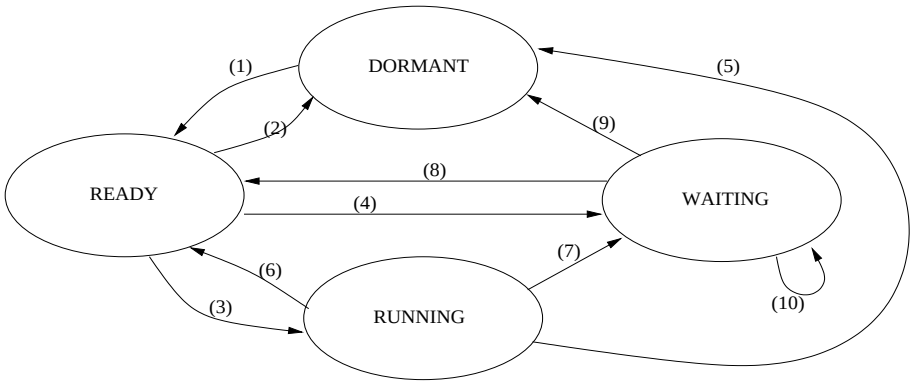



Figure 24: ARINC process state diagram

`Transition (4)`: the process is suspended by another process within the partition.
`Transition (5)`: the process stops itself.
`Transition (6)`: the process waits on a TIMED_DELAY of zero or is preempted by another process within the partition.
`Transition (7)`: the process suspends itself, or the process attempts to access a resource (delay, semaphore, period, event, message) which is not currently available and the process accepts to wait.
`Transition (8)`: the process is resumed, or the resource the process was waiting for becomes available or the time-out expires.
`Transition (9)`: the process is stopped by another process within the partition.
`Transition (10)`: a process already waiting to access a resource (delay, semaphore, period, event, message) is suspended, or a process which is both waiting to access a resource and is suspended, is either resumed, or the resource becomes available, or the time-out expires.

### 8.2.2 Types

The following types are particular to the process management services:

| **Types** | **Nature** |
|---|---|
| ProcessID_type | numeric |
| ProcessName_type | string |
| Priority_type | numeric |
| StackSize_type | numeric |
| Locklevel_type | numeric |
| SystemAddress_type | implementation dependant |
| ProcessState_type | (DORMANT, READY, RUNNING, WAITING) |
| Deadline_type | (SOFT, HARD) |
| SystemTime_type | implementation dependant |
| ProcessAttributes_type | struct ( ProcessName_type Name;<br>SystemAddress_type EntryPt;<br>StackSize_type StackSize;<br>Priority_type BasePriority;<br>SystemTime_type Period;<br>SystemTime_type TimeCapacity;<br>Deadline_type Deadline;) |
| ProcessStatus_type | struct ( ProcessAttributes_type ProcessAttributes;<br>Priority_type CurrentPriority;<br>Deadline_typeDeadlineTime;<br>ProcessState_type ProcessState;) |

### 8.2.3 Process descriptor

This section contains process descriptor access services.

- process PROCESS_CHECKID
  Interface:

| **Label** | **Nature** | **Type** | **Comments** |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *present* | output | boolean | presence indicator |
| *status* | output | ProcessStatus_type | process status |

  Processing:
  When there is a process identified by *process_ID* in the partition, the presence indicator is `TRUE` and *status* contains the corresponding process status informations. Otherwise, *present* is `FALSE`.

```
process PROCESS_CHECKID =
     ( ? ProcessID_type process_ID;
       ! boolean present;
         ProcessStatus_type status;
     )
     spec (| (| process_ID -> present
              | { process_ID -> status } when present
              |)
           | (| process_ID ^= present
              | status ^= when present
              |)
           |)
;
```

- process PROCESS_CHECKAPERIODIC
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *aperiodic* | output | boolean | periodicity indicator |

  Processing:
  When *process_ID* identifies some aperiodic process in the partition, *aperiodic* is `TRUE`. Otherwise, *aperiodic* is `FALSE`.

```
process PROCESS_CHECKAPERIODIC =
     ( ? ProcessID_type process_ID;
       ! boolean aperiodic;
     )
     spec (| (| process_ID -> aperiodic |)
           | (| process_ID ^= aperiodic |)
           |)
;
```

- process PROCESS_CHECKERRORHANDLER
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *err_handler* | output | boolean | *true* if error handler, else *false* |

  Processing:
  When *process_ID* identifies the error handler process in the partition, *err_handler* is `TRUE`. Otherwise, *err_handler* is `FALSE`.

```
process PROCESS_CHECKERRORHANDLER =
     ( ? ProcessID_type process_ID;
       ! boolean err_handler;
     )
     spec (| (| process_ID -> err_handler |)
           | (| process_ID ^= err_handler |)
           |)
;
```

- process PROCESS_SETCREATED
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |

Processing:
The creation attribute associated with the process identified by *process_ID* is set to `TRUE`.

```
process PROCESS_SETCREATED =
     ( ? ProcessID_type process_ID;
     )
```

- process PROCESS_CHECKCREATED
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_name* | input | ProcessName_type | process name |
| *created* | output | boolean | creation indicator |

Processing:
When the process named *process_name* is already created, the output *created* is `TRUE`. Otherwise, *created* is `FALSE`.

```
process PROCESS_CHECKCREATED =
      ( ? ProcessName_type process_name;
        ! boolean created;
      )
      spec (| (| process_name -> created |)
            | (| process_name ^= created |)
            |)
;
```

- process PROCESS_GETPERIOD
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *period* | output | SystemTime_type | process period |

Processing:
The output *period* is the period associated with the process identified by *process_ID*.

```
process PROCESS_GETPERIOD =
     ( ? ProcessID_type process_ID;
       ! SystemTime_type period;
     )
     spec (| (| process_ID -> period |)
           | (| process_ID ^= period |)
           |)
;
```

- process PROCESS_RESETSTACK
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |

  Processing:
  This service resets the stack of the process identified by *process_ID*.

```
process PROCESS_RESETSTACK =
     ( ? ProcessID_type process_ID;
     )
;
```

- process PROCESS_REMOVEWAITINGQUEUE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |

  Processing:
  This service removes the process identified by *process_ID* from any waiting queue where it may be listed.

```
process PROCESS_REMOVEWAITINGQUEUE =
     ( ? ProcessID_type process_ID;
     )
;
```

- process PROCESS_PREVIOUSPREEMPTED
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *preempted* | output | boolean | answer to the request |

  Processing:
  When *process_ID* identifies the previous preempted process, the output *preempted* is `TRUE`. Otherwise, *preempted* is `FALSE`.

```
process PROCESS_PREVIOUSPREEMPTED =
     ( ? ProcessID_type process_ID;
       ! boolean preempted;
     )
     spec (| (| process_ID -> preempted |)
           | (| process_ID ^= preempted |)
           |)
;
```

- process PROCESS_CHECKOPERATINGMODE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *op_mode* | output | OperatingMode_type | operating mode |

  Processing:
  The output *op_mode* indicates the current operating mode of the partition.

```
process PROCESS_CHECKOPERATINGMODE =
     ( ! OperatingMode_type op_mode;
     )
;
```

- process PROCESS_CHECKPREEMPTION
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *req* | input | event | service request |
| *enabled* | output | boolean | preemption status |

  Processing:
  On a request of this service (*req* is present); when the partition enables preemption, the output *enabled* is `TRUE`. Otherwise, *enabled* is `FALSE`.

```
process PROCESS_CHECKPREEMPTION =
     ( ? event req;
       ! boolean enabled;
     )
     spec (| (| req -> enabled |)
           | (| req ^= enabled |)
           |)
;
```

- process PROCESS_CHECKWAITINDICATOR
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *wait_ind* | output | boolean | waiting indicator |

  Processing:
  When the process identified by *process_ID* is waiting for some resource (semaphore, buffer...), the output *wait_ind* is `TRUE`. Otherwise, *wait_ind* is `FALSE`.

```
process PROCESS_CHECKWAITINDICATOR =
     ( ? ProcessID_type process_ID;
       ! boolean wait_ind;
```

```
      )
      spec (| (| process_ID -> wait_ind |)
            | (| process_ID ^= wait_ind |)
            |)
;
```

- process PROCESS_GETSTATE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *state* | output | ProcessState_type | process state |

  Processing:
  The output *state* is the current state of the process identified by *process_ID*.

```
process PROCESS_GETSTATE =
      ( ? ProcessID_type process_ID;
        ! ProcessState_type state;
      )
      spec (| (| process_ID -> state |)
            | (| process_ID ^= state |)
            |)
;
```

- process PROCESS_GETACTIVESTATUS
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | output | ProcessID_type | process identifier |
| *status* | output | ProcessStatus_type | process status |

  Processing:
  The output *process_ID* identifies the current active process in the partition, and *status* is its associated status informations.

```
process PROCESS_GETACTIVESTATUS =
      ( ! ProcessID_type process_ID;
          ProcessStatus_type status;
      )
      spec (| process_ID ^= status |)
;
```

- process PROCESS_GETBASICPRIORITY
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *basic_priority* | output | Priority_type | process basic priority |

Processing:
The output *basic_ priority* is the basic priority of the process identified by *process_ ID*.

```
process PROCESS_GETBASICPRIORITY =
      ( ? ProcessID_type process_ID;
        ! Priority_type basic_priority;
      )
      spec (| (| process_ID -> basic_priority |)
            | (| process_ID ^= basic_priority |)
            |)
 ;
```

- process PROCESS_GETTIMECAPACITY
Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ ID* | input | ProcessID_type | process identifier |
| *time_ capacity* | output | SystemTime_type | process time capacity |

Processing:
The output *time_ capacity* is the time capacity of the process identified by *process_ ID*.

```
process PROCESS_GETTIMECAPACITY =
      ( ? ProcessID_type process_ID;
        ! SystemTime_type time_capacity;
      )
      spec (| (| process_ID -> time_capacity |)
            | (| process_ID ^= time_capacity |)
            |)
 ;
```

- process PROCESS_RECORD
Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *name* | input | ProcessName_type | process name |
| *entry_ point* | input | SystemAddress_type | entry point |
| *stack_ size* | input | StackSize_type | stack size |
| *base_ priority* | input | Priority_type | basic priority |
| *period* | input | SystemTime_type | period |
| *time_ capacity* | input | SystemTime_type | time capacity |
| *deadline* | input | Deadline_type | deadline indicator |
| *process_ ID* | output | ProcessID_type | process identifier |

Processing:
This service allocates a new process descriptor if there is enough space in the process descriptor manager, and initializes the descriptor with the input parameters. The output *process_ ID* is the returned process identifier.

```
process PROCESS_RECORD =
     ( ? ProcessName_type name;
         SystemAddress_type entry_point;
         StackSize_type stack_size;
         Priority_type base_priority;
         SystemTime_type period;
         SystemTime_type time_capacity;
         Deadline_type deadline;
       ! ProcessID_type process_ID;
     )
     spec (| (| { { name, entry_point, stack_size, base_priority,
                    period, time_capacity, deadline } -> process_ID }
                    when C_enough_space
             |)
           | (| name ^= entry_point ^= stack_size ^= base_priority
                ^= period ^= time_capacity ^= deadline ^= C_enough_space
             | process_ID ^= when C_enough_space
             |)
     where boolean C_enough_space;
;
```

- process PROCESS_RELEASERESOURCES
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |

  Processing:
  This service releases all the resources detained by the process identified by *process_ID*.

```
process PROCESS_RELEASERESOURCES =
     ( ? ProcessID_type process_ID;
     )
;
```

- process PROCESS_RESETCONTEXT
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |

  Processing:
  This service resets the context of the process identified by *process_ID*.

```
process PROCESS_RESETCONTEXT =
     ( ? ProcessID_type process_ID;
     )
;
```

- process PROCESS_RESETERRORSTATUS
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |

  Processing:
  This service resets the error status of the process identified by *process_ID*.

```
process PROCESS_RESETERRORSTATUS =
     ( ? event reset_req;
     )
;
```

- process PROCESS_RESETLOCKLEVEL
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *reset_req* | input | event | service request |

  Processing:
  A request of this service (*reset_req* is present) resets the lock level of the partition.

```
process PROCESS_RESETLOCKLEVEL =
     ( ? event reset_req;
     )
;
```

- process PROCESS_SCHEDULINGREQUEST
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *sched_req* | input | event | rescheduling request |
| *diagnostic* | output | boolean | diagnostic of the request |

  Processing:
  On a request of this service (*sched_req* is present), a process rescheduling is tried to be performed. When the lock level of the partition indicates "unlocked", the output *diagnostic* is `TRUE`. Otherwise, *diagnostic* is `FALSE`.

```
process PROCESS_SCHEDULINGREQUEST =
     ( ? event sched_req;
       ! boolean diagnostic;
     )
     spec (| (| sched_req -> diagnostic |)
           | (| sched_req ^= diagnostic |)
           |)
;
```

- process PROCESS_SETALLSTATE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *range* | input | array of Process_type | set of processes |
| *state* | input | ProcessState_type | state value |

  Processing:
  For each process whose identifier appears in *range*, its state is set to input *state* value.

```
process PROCESS_SETALLSTATE =
     ( ? [MAX_NUMBER_OF_PROCESSES]ProcessID_type range;
         ProcessState_type state;
     )
     spec (| range ^= state |)
;
```

- process PROCESS_SETATTRIBUTES
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *attributes* | input | ProcessAttributes_type | process attributes |

  Processing:
  The attributes of the process identified by *process_ID* are set to *attributes* value.

```
process PROCESS_SETATTRIBUTES =
     ( ? ProcessID_type process_ID;
         ProcessAttributes_type attributes;
     )
     spec (| process_ID ^= attributes |)
;
```

- process PROCESS_SETDEADLINETIME
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *deadline* | input | SystemTime_type | deadline time |

  Processing:
  The deadline time attribute of the process identified by *process_ID* is set to *deadline* value.

```
process PROCESS_SETDEADLINETIME =
     ( ? ProcessID_type process_ID;
         SystemTime_type deadline;
```

```
        )
        spec (| process_ID ^= deadline |)
    ;
```

- process PROCESS_SETSTATE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *state* | input | ProcessState_type | state value |

  Processing:
  The state attribute of the process identified by *process_ID* is set to *state* value.

```
process PROCESS_SETSTATE =
     ( ? ProcessID_type process_ID;
         ProcessState_type state;
     )
     spec (| process_ID ^= state |)
;
```

- process PROCESS_CREATEMANAGER
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *manager_size* | input | integer | maximum size of the manager |

  Processing:
  This service creates a process descriptor manager which can contain at most *manager_size* descriptors.

```
process PROCESS_MANAGERCREATE =
     ( ? integer manager_size;
     )
```

- process PROCESS_COPYMESSAGE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *message* | input | Message_type | message identifier |
| *process_ID* | input | ProcessID_type | process identifier |

  Processing:
  This service copies the message denoted by *message* in a special location associated with the process identified by *process_ID*, which has been blocked on a receive message request, and now released.

```
process PROCESS_COPYMESSAGE =
     ( ? Message_type message;
         ProcessID_type process_ID;
     )
    spec (| message ^= process_ID |)
;
```

- process PROCESS_RETRIEVEMESSAGE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *message_OUT* | output | Message_type | message |

  Processing:

  This service allows a process (identified by *process_ID*) blocked on a receive message request to retrieve a message as soon as it becomes available. The output *message_OUT* is the retrieved message.

```
process PROCESS_RETRIEVEMESSAGE =
     ( ? ProcessID_type process_ID;
       ! Message_type message_OUT;
     )
     spec (| (| message_OUT -> process_ID |)
           | (| message_OUT ^= process_ID |)
           |)
;
```

### 8.2.4 Process waiting queues

Here, we present process queuing services (e.g. insert (resp. remove) a process in (resp. from) a queue). They allow to put a process in a waiting queue associated with a resource when this resource is not available.

- process INSERT_BLACKBOARDQUEUE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *blackboard_IN* | input | APEX_Blackboard_type | blackboard |

  Processing:
  This service inserts the process identified by *process_ID* in the process waiting queue associated with the blackboard *blackboard_IN*.

```
process INSERT_BLACKBOARDQUEUE =
     ( ? ProcessID_type process_ID;
         APEX_Blackboard_type blackboard_IN;
     )
     spec (| process_ID ^= blackboard_IN |)
;
```

- process INSERT_BUFFERRECEIVEQUEUE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *buffer_IN* | input | APEX_Buffer_type | buffer |

  Processing:
  This service inserts the process identified by *process_ID* in the process waiting queue, associated with processes that are waiting for receiving a message from the buffer *buffer_IN*.

```
process INSERT_BUFFERRECEIVEQUEUE =
     ( ? ProcessID_type process_ID;
         APEX_Buffer_type buffer_IN;
     )
     spec (| process_ID ^= buffer_IN |)
;
```

- process INSERT_BUFFERSENDQUEUE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *message* | input | Message_type | message |
| *buffer_IN* | input | APEX_Buffer_type | buffer |

Processing:
This service inserts the couple (*process_ID*, *message*) in the process waiting queue, associated with processes that are waiting for sending a message in the buffer *buffer_IN*.

```
process INSERT_BUFFERSENDQUEUE =
     ( ? ProcessID_type process_ID;
         Message_type message;
         APEX_Buffer_type buffer_IN;
     )
     spec (| process_ID ^= message ^= buffer_IN |)
```

- process INSERT_EVENTQUEUE
Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *event_IN* | input | APEX_Event_type | event |

Processing:
This service inserts the process identified by *process_ID* in the process waiting queue associated with the event *event_IN*.

```
process INSERT_EVENTQUEUE =
     ( ? ProcessID_type process_ID;
         APEX_Event_type event_IN;
     )
     spec (| process_ID ^= event_IN |)
;
```

- process INSERT_SEMAPHOREQUEUE
Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *semaphore_IN* | input | APEX_Semaphore_type | semaphore |

Processing:
This service inserts the process identified by *process_ID* in the process waiting queue associated with the semaphore *semaphore_IN*.

```
process INSERT_SEMAPHOREQUEUE =
     ( ? ProcessID_type process_ID;
         APEX_Semaphore_type semaphore_IN;
     )
     spec (| process_ID ^= semaphore_IN |)
;
```

- process INSERT_QUEUINGPORTRECEIVEQUEUE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *queuingPort_IN* | input | APEX_QueuingPort_type | queuing port |

  Processing:
  This service inserts the process identified by *process_ID* in the process waiting queue, associated with processes that are waiting for receiving a message from the queuing port *queuingPort_IN*.

```
process INSERT_QUEUINGPORTRECEIVEQUEUE =
     ( ? ProcessID_type process_ID;
         APEX_QueuingPort_type queuingPort_IN;
     )
     spec (| process_ID ^= queuingPort_IN |)
```

- process INSERT_QUEUINGPORTSENDQUEUE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *message* | input | Message_type | message |
| *queuingPort_IN* | input | APEX_QueuingPort_type | queuing port |

  Processing:
  This service inserts the couple (*process_ID*, *message*) in the process waiting queue, associated with processes that are waiting for sending a message in the queuing port *queuingPort_IN*.

```
process INSERT_QUEUINGPORTSENDQUEUE =
     ( ? ProcessID_type process_ID;
         Message_type message;
         APEX_QueuingPort_type queuingPort_IN;
     )
     spec (| process_ID ^= message ^= queuingPort_IN |)
;
```

- process REMOVE_BUFFERRECEIVEQUEUE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *buffer_IN* | input | APEX_Buffer_type | buffer |
| *process_ID* | output | ProcessID_type | process identifier |

  Processing:
  This service removes a process identified by *process_ID* from the process waiting queue, associated with processes that are waiting for receiving a message from *buffer_IN*.

```
process REMOVE_BUFFERRECEIVEQUEUE =
     ( ? APEX_Buffer_type buffer_IN;
       ! ProcessID_type process_ID;
     )
     spec (| (| buffer_IN -> process_ID |)
           | (| process_ID ^= buffer_IN |)
           |)
;
```

- process REMOVE_BUFFERSENDQUEUE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *buffer_IN* | input | APEX_Buffer_type | buffer |
| *process_ID* | output | ProcessID_type | process identifier |
| *message* | output | Message_type | message |

  Processing:
  This service removes the couple (*process_ID*, *message*) from the process waiting queue, associated with processes that are waiting for sending a message in *buffer_IN*.

```
process REMOVE_BUFFERSENDQUEUE =
     ( ? APEX_Buffer_type buffer_IN;
       ! ProcessID_type process_ID;
         Message_type message;
     )
     spec (| (| buffer_IN -> { process_ID, message } |)
           | (| process_ID ^= message ^= buffer_IN |)
           |)
;
```

- process REMOVE_SEMAPHOREQUEUE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *semaphore_IN* | input | APEX_Semaphore_type | semaphore |
| *process_ID* | output | ProcessID_type | process identifier |

  Processing:
  This service removes a process identified by *process_ID* from the process waiting queue associated with the semaphore *semaphore_IN*.

```
process REMOVE_SEMAPHOREQUEUE =
     ( ? APEX_Semaphore_type semaphore_IN;
       ! ProcessID_type process_ID;
     )
     spec (| (| semaphore_IN -> process_ID |)
           | (| process_ID ^= semaphore_IN |)
```

```
            |)
;
```

- process REMOVALL_BLACKBOARDQUEUE
  Interface:

| **Label** | **Nature** | **Type** | **Comments** |
|---|---|---|---|
| *blackboard_IN* | input | APEX_Blackboard_type | blackboard |
| *process_range* | output | ProcessID_type | process range |

  Processing:
  This service removes all the processes present in the process waiting queue associated with the blackboard *blackboard_IN*. The output *process_range* contains these process identifiers.

```
process REMOVALL_BLACKBOARDQUEUE =
     ( ? APEX_Blackboard_type blackboard_IN;
       ! [MAX_NUMBER_OF_PROCESSES]ProcessID_type process_range;
     )
     spec (| (| blackboard_IN -> process_range |)
           | (| process_range ^= blackboard_IN |)
           |)
;
```

- process REMOVALL_EVENTQUEUE
  Interface:

| **Label** | **Nature** | **Type** | **Comments** |
|---|---|---|---|
| *event_IN* | input | APEX_Event_type | event |
| *process_range* | output | ProcessID_type | process range |

  Processing:
  This service removes all the processes present in the process waiting queue associated with the event *event_IN*. The output *process_range* contains these process identifiers.

```
process REMOVALL_EVENTQUEUE =
     ( ? APEX_Event_type event_IN;
       ! [MAX_NUMBER_OF_PROCESSES]ProcessID_type process_range;
     )
     spec (| (| event_IN -> process_range |)
           | (| process_range ^= event_IN |)
           |)
;
```

- process REMOVE_QUEUINGPORTRECEIVEQUEUE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *queuingPort_IN* | input | APEX_QueuingPort_type | queuing port |
| *process_ID* | output | ProcessID_type | process identifier |

  Processing:
  This service removes the process identified by *process_ID* from the process waiting queue, associated with processes that are waiting for receiving a message from *queuingPort_IN*.

```
process REMOVE_QUEUINGPORTRECEIVEQUEUE =
     ( ? APEX_QueuingPort_type queuingPort_IN;
       ! ProcessID_type process_ID;
     )
     spec (| (| queuingPort_IN -> process_ID |)
           | (| process_ID ^= queuingPort_IN |)
           |)
;
```

- process REMOVE_QUEUINGPORTSENDQUEUE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *queuingPort_IN* | input | APEX_QueuingPort_type | queuing port |
| *process_ID* | output | ProcessID_type | process identifier |
| *message* | output | Message_type | message |

  Processing:
  This service removes the couple (*process_ID*, *message*) from the process waiting queue, associated with processes that are waiting for sending a message in *queuingPort_IN*.

```
process REMOVE_QUEUINGPORTSENDQUEUE =
     ( ? APEX_QueuingPort_type queuingPort_IN;
       ! ProcessID_type process_ID;
         Message_type message;
     )
     spec (| (| queuingPort_IN -> { process_ID, message } |)
           | (| process_ID ^= message ^= queuingPort_IN |)
           |)
;
```

### 8.2.5 Process management

Now, we give information about the interface and the functionnalitiy of the process management services.

- process GET_PROCESS_ID*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_name* | input | ProcessName_type | process name |
| *process_ID* | output | ProcessID_type | its identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service allows a process to obtain a process identifier by specifying the process name. The return code indicates that the request has been issued.

  1. The return code value is:
     - `INVALID_CONFIG` when no process is named *process_name* in the current partition;
     - `NO_ERROR` otherwise.
  2. In the case of successful completion, *process_ID* denotes the identifier of the process named *process_name*.

```
process GET_PROCESS_ID =
     ( ? ProcessName_type process_name;
       ! ProcessID_type process_ID;
         ReturnCode_type return_code;
     )
     spec (| (| process_name -> return_code
              | { process_name -> process_ID } when (return_code = #NO_ERROR)
              |)
           | (| process_name ^= return_code
              | process_ID ^= when (return_code = #NO_ERROR)
              |)
           |)
;
```

- process GET_PROCESS_STATUS*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *process_status* | output | ProcStatus_type | its current status |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service allows to get the current status of the specified process. Here also, the return code indicates that the request has been issued.

1. The return code value is:
   - INVALID_PARAM when no process is identified by *process_ID* in the current partition;
   - NO_ERROR otherwise.
2. In the case of successful completion, *process_status* denotes the current status of the specified process.

```
process GET_PROCESS_STATUS =
     ( ? ProcessID_type process_ID;
       ! ProcessStatus_type process_status;
         ReturnCode_type return_code;
     )
     spec (| (| process_ID -> return_code
              | { process_ID -> process_status } when (return_code = #NO_ERROR)
              |)
           | (| process_ID ^= return_code
              | process_status ^= when (return_code = #NO_ERROR)
              |)
           |)
;
```

- process CREATE_PROCESS*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *attributes* | input | ProcessAttributes_type | process attributes |
| *process_ID* | output | ProcessID_type | process identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to create a process and returns an identifier that denotes the created process. Creation does not imply dynamic memory allocation, it only creates a link between the given name and a statically allocated process with a suitable stack area having the same name.

  1. The return code value is:
     - INVALID_CONFIG when there is no process named *attributes.Name* in the partition, or at least one of *attributes.entry_point*, *attributes.stack_size*, *attributes.base_priority*, *attributes.period* and *attributes.time_capacity* is out of range.
     - NO_ACTION when a process with the same name is already created;
     - INVALID_MODE when the operating mode is NORMAL;
     - NO_ERROR otherwise.
  2. In the case of successful completion, the process attributes are set to *attributes*. The process state is DORMANT. The context and stack are reset. Finally, the output *process_ID* denotes the identifier of the created process.

```
process CREATE_PROCESS =
     ( ? ProcessAttributes_type attributes;
       ! ProcessID_type process_ID;
         ReturnCode_type return_code;
     )
     spec(| (| attributes -> return_code
             | { attributes -> process_ID } when (return_code = #NO_ERROR)
             |)
          | (| return_code ^= attributes
             | process_ID ^= when (return_code = #NO_ERROR)
             |)
          |)
;
```

- process SET_PRIORITY*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *priority* | input | Priority_type | its priority |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to change the process current priority.

  1. The return code value is:
     - `INVALID_PARAM` when there is no process identified by *process_ID*, or *priority* is out of range;
     - `INVALID_MODE` when the specified process state is `DORMANT`;
     - `NO_ERROR` otherwise.
  2. The specified process priority is set to *priority*. Process scheduling is performed if preemption is enabled.

```
process SET_PRIORITY =
     ( ? ProcessID_type process_ID;
         Priority_type priority;
       ! ReturnCode_type return_code;
     )
     spec (| (| { process_ID, priority } -> return_code |)
           | (| return_code ^= process_ID ^= priority |)
           |)
;
```

- process SUSPEND_SELF*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *timeout* | input | SystemTime_type | delay of suspension |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service is used to suspend the execution of the current process if aperiodic, until the RESUME service request is issued or the specified time-out value expires.

1. The return code value is:
   - `INVALID_PARAM` when *timeout* is out of range;
   - `INVALID_MODE` when preemption is disabled or the process is an error handler, or the process is periodic;
   - `NO_ERROR` otherwise.
2. In the case of successful completion,
   - when *timeout* is zero, no other action is performed;
   - otherwise, the current process state is set to `WAITING`. If *timeout* is not infinite, a time counter is initiated with duration *timeout*. Process scheduling is performed.

```
process SUSPEND_SELF =
     ( ? SystemTime_type timeout;
       ! ReturnCode_type return_code;
     )
     spec (| (| timeout -> return_code |)
           | (| timeout ^= return_code |)
           |)
;
```

- process SUSPEND*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service allows the current process to suspend any aperiodic process, identified by *process_ID* except itself.

1. The return code value is:
   - `INVALID_PARAM` when *process_ID* does not identify any process or identifies itself;
   - `INVALID_MODE` when the state of the specified process is `DORMANT`, or specified process is periodic;
   - `NO_ERROR` otherwise.
2. In the case of successful completion, the state of the specified process is set to `WAITING`.

```
process SUSPEND =
     ( ? ProcessID_type process_ID;
       ! ReturnCode_type return_code;
```

```
      )
      spec (| (| process_ID -> return_code |)
            | (| process_ID ^= return_code |)
            |)
;
```

- process RESUME*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to resume a process previously suspended.

  1. The return code value is:
     - `INVALID_PARAM` when *process_ID* does not identify any process or identifies itself;
     - `INVALID_MODE` when the state of the specified process is `DORMANT`;
     - `NO_ERROR` otherwise.
  2. In the case of successful completion, if the specified process is suspended with a time-out, the affected time counter is stopped. If the specified process is not waiting on a resource, its state is set to `READY`. Process scheduling is performed if preemption is enabled.

```
process RESUME =
     ( ? ProcessID_type process_ID;
       ! ReturnCode_type return_code;
     )
     spec (| (| process_ID -> return_code |)
           | (| process_ID ^= return_code |)
           |)
;
```

- process STOP_SELF*
  Interface:
  No input and output. Processing:

  This service is used to stop the current process. If the current process is not the error handler process, the partition is placed in the unlocked condition.

  1. All the resources used by the current process are released.
  2. If process is not an error handler, the lock level counter is reset.
  3. The process state is set to `DORMANT`.

4. When the current process is an error handler and preemption is disabled, return to the previous process (reset the locklevel and ask for rescheduling), otherwise ask only process scheduling.

```
process STOP_SELF =
     ( )
;
```

- process STOP*

Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service is used to abort the execution of any process except the executing one.

1. The return code value is:
   - `INVALID_PARAM` when *process_ID* does not identify any process or identifies itself;
   - `NO_ACTION` when the state of the specified process is `DORMANT`;
   - `NO_ERROR` otherwise.
2. In the case of successful completion, the specified process state is set to `DORMANT`. All the resources used by this process are released. If the current process is an error handler, and the specified process is preempted by an error handler, the lock level counter is reset. If the specified process is pending in a queue, it will be removed.

```
process STOP =
     ( ? ProcessID_type process_ID;
       ! ReturnCode_type return_code;
     )
     spec (| (| process_ID -> return_code |)
           | (| process_ID ^= return_code |)
           |)
;
```

- process START*

Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service is used to start the execution of a process.

1. The return code value is:
   - `INVALID_PARAM` when *process_ID* does not identify any process;
   - `NO_ACTION` when the state of the specified process is not `DORMANT`;
   - `NO_ERROR` otherwise.
2. In the case of successful completion, the current priority of the specified process is set to its base priority. Context and stack are reset. The specified process state is set to `READY`.
   - If operating mode is NORMAL, a new deadline time value is set for the specified process, error status is reset, and process scheduling is performed if preemption is enabled.
   - Else the start of the process will be effective, and its deadline time value will be calculated at the end of the initialization phase (when the partition mode becomes NORMAL).

```
process START =
     ( ? ProcessID_type process_ID;
       ! ReturnCode_type return_code;
     )
     spec (| (| process_ID -> return_code |)
           | (| process_ID ^= return_code |)
           |)
;
```

- process LOCK_PREEMPTION*

Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *lock_level* | input | LockLevel_type | lock level |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service is used to increment the lock level of the partition and disable process rescheduling for a partition.

1. The return code value is:
   - `INVALID_CONFIG` when *lock_level* is greater than the maximum value of the lock level;
   - `NO_ERROR` otherwise.
2. In the case of successful completion, the lock level is incremented by one.

```
process LOCK_PREEMPTION =
     ( ? LockLevel_type lock_level;
       ! ReturnCode_type return_code;
     )
```

```
      spec (| (| lock_level -> return_code |)
            | (| lock_level ^= return_code |)
            |)
;
```

- process UNLOCK_PREEMPTION*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *lock_level* | input | LockLevel_type | lock level |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to decrement the lock level of the partition.

  1. The return code value is:
     - `NO_ACTION` when the lock level indicates unlocked;
     - `NO_ERROR` otherwise.
  2. In the case of successful completion, the lock level is decremented by one. If it becomes zero, process scheduling is performed.

```
process UNLOCK_PREEMPTION =
     ( ? LockLevel_type lock_level;
       ! ReturnCode_type return_code;
     )
     spec (| (| lock_level -> return_code |)
           | (| lock_level ^= return_code |)
           |)
;
```

## 8.3 Intra-partition communication and synchronization mechanisms

We first remind the main intra-partition communication and synchronization mechanisms. Then, we explain about the associated services.

1. **Blackboard** is used to display and read messages. No message queues are allowed. Any message displayed on a blackboard remains there until the message is either cleared or overwritten by a new instance of the message. This allows sending processes to display messages at any time, and receiving processes to access the latest message at any time. A process can read a message from a blackboard, display a message on a blackboard or clear a blackboard. Rescheduling of processes will occur when a process attempts to read a message from an empty blackboard.
2. **Buffer** is used to send and receive messages. Buffer is allowed to store multiple messages in message queues. A message sent by the sending process is stored in the message queue in FIFO order. No message should be lost in this queuing mode. The number of messages that can be stored in a buffer is determined by the size of the buffer, and is specified at creation time. A process can send a message to a buffer or receive a message from a buffer. Rescheduling of processes will occur when a process attempts either to receive a message from an empty buffer or to send a message to a full buffer.
3. **Event** is a communication mechanism which permits notification of an occurrence of a condition to processes which may wait for it. An event is composed of a bi-valued state variable (states called "up" and "down") and a set of waiting processes (initially empty). A process can set and reset events and also wait on events that are created in the same partition. Rescheduling of processes will occur when a process attempts to wait on an event which is "down" (reset before by another process or during initialization), and when a process sets an event "up".
4. **Semaphore** is used for synchronization. The semaphore defined here is a counting one and is commonly used to provide controlled access to partition resources. A process waits for a semaphore to gain access to the resource, and then signals the semaphore after it releases the resource. A semaphore's value indicates the number of currently available resources. Rescheduling of processes will occur when a process attempts to wait on a zero value semaphore, and when a semaphore is signaled that has processes queued on it.

### 8.3.1 Types

Here also, we give information about the types used by the services.

The following types are related to process queuing on a service request when the specified resource is not available.

| **Label** | **Nature** |
|---|---|
| QueuingDiscipline_type | enum (FIFO, PRIORITY) |
| QueueAddress_type | external |

The next types define the communication and synchronization mechanisms.

| Label | Nature |
|---|---|
| MessageSize_type | numeric |
| MessageArea_type | implementation dependant |
| Message_type | struct ( MessageArea_type Message_Area;<br>MessageSize_type Message_Size;) |
| Comm_ComponentID_type | numeric |
| Comm_ComponentName_type | string |
| APEX_Blackboard_type | struct ( ComponentName_type Blackboard_Name;<br>MessageID_type Message_ID;<br>boolean Empty_Indicator;<br>integer Waiting_Processes;<br>QueueAddress_type Waiting_Queue;) |
| BlackboardStatus_type | struct ( boolean Empty_Indicator;<br>integer Waiting_Processes;) |
| APEX_Buffer_type | struct ( ComponentName_type Buffer_Name;<br>[MAX_SIZE]MessageID_type Message_Range;<br>integer Nb_Message;<br>integer Max_Message;<br>integer WaitingOnSend_Processes;<br>integer WaitingOnReceive_Processes;<br>QueuingDiscipline_type Queuing_Discipline;<br>QueueAddress_type WaitingOnSend_Queue;<br>QueueAddress_type WaitingOnReceive_Queue;) |
| BufferStatus_type | struct ( integer Nb_Message;<br>integer Max_Message;<br>integer Waiting_Processes;) |
| APEX_Event_type | struct ( ComponentName_type Event_Name;<br>boolean Event_State;<br>integer Waiting_Processes;<br>QueueAddress_type Waiting_Queue;) |
| EventStatus_type | struct ( boolean Event_State;<br>integer Waiting_Processes;) |
| APEX_Semaphore_type | struct ( ComponentName_type Semaphore_Name;<br>integer Current_Value;<br>integer Maximum_Value;<br>integer Waiting_Processes;<br>QueuingDiscipline_type Queuing_Discipline;<br>QueueAddress_type Waiting_Queue;) |
| SemaphoreStatus_type | struct ( integer Current_Value;<br>integer Maximum_Value;<br>integer Waiting_Processes;) |

### 8.3.2 Intra-partition communication and synchronization mechanisms manager

Every communication and synchronization mechanism is uniquely identified within the partition. The identifier is used to refer to the mechanisms in their corresponding managers. This section presents the mechanisms management services.

- process BLACKBOARD_CHECKID
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *blackboard_ID* | input | Comm_ComponentID_type | blackboard identifier |
| *present* | output | boolean | presence indicator |
| *blackboard_OUT* | output | APEX_Blackboard_type | blackboard |

  Processing:
  When there is a blackboard identified by *blackboard_ID*, the presence indicator is `TRUE` and *blackboard_OUT* is this blackboard. Otherwise, *present* is `FALSE`.

```
process BLACKBOARD_CHECKID =
     ( ? Comm_ComponentID_type blackboard_ID;
       ! boolean present;
         APEX_Blackboard_type blackboard_OUT;
     )
     spec (| (| blackboard_ID -> present
              | { blackboard_ID -> blackboard_OUT } when present
              |)
           | (| blackboard_ID ^= present
              | blackboard_OUT ^= when present
              |)
           |)
;
```

- process BLACKBOARD_CHECKNAME
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *blackboard_name* | input | Comm_ComponentName_type | blackboard name |
| *present* | output | boolean | presence indicator |
| *blackboard_ID* | output | Comm_ComponentID_type | blackboard identifier |

  Processing:
  When there is a blackboard named *blackboard_name*, the presence indicator is `TRUE` and *blackboard_ID* is the corresponding identifier. Otherwise, *present* is `FALSE`.

```
process BLACKBOARD_CHECKNAME =
    ( ? Comm_ComponentName_type blackboard_name;
      ! boolean present;
        Comm_ComponentID_type blackboard_ID;
    )
    spec (| (| blackboard_name -> present
             | { blackboard_name -> blackboard_ID } when present
             |)
          | (| blackboard_name ^= present
             | blackboard_ID ^= when present
             |)
          |)
;
```

- process BLACKBOARD_RECORD
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *blackboard_name* | input | Comm_ComponentName_type | blackboard name |
| *message_size* | input | integer | message size |
| *blackboard_ID* | output | Comm_ComponentID_type | blackboard identifier |
| *recorded* | output | boolean | record indicator |

  Processing:
  This service records the blackboard named *blackboard_name*. The size of the messages displayed on this blackboard is *message_size*. The output *blackboard_ID* is the identifier associated with the blackboard. The record indicator indicates whether or not the blackboard has been actually recorded (that means there was enough space in the manager to define a new blackboard). So, it carries the value `TRUE` when the blackboard is recorded; otherwise, *recorded* is `FALSE`.

```
process BLACKBOARD_RECORD =
     ( ? Comm_ComponentName_type blackboard_name;
         MessageSize_type message_size;
       ! Comm_ComponentID_type blackboard_ID;
         boolean recorded;
     )
     spec (| (| { blackboard_name,message_size } -> recorded
              | { { blackboard_name,message_size } ->
                            blackboard_ID } when recorded
              |)
           | (| blackboard_name ^= message_size ^= recorded
             | blackboard_ID ^= when recorded
             |)
          |)
;
```

- process BLACKBOARD_UPDATE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *blackboard_IN* | input | APEX_Blackboard_type | blackboard |

  Processing:
  This service updates the blackboard named *blackboard_IN*.Name in the manager with the status of *blackboard_IN*.

```
process BLACKBOARD_UPDATE =
     ( ? APEX_Blackboard_type blackboard_IN;
     )
;
```

- process BUFFER_CHECKID
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *buffer_ID* | input | Comm_ComponentID_type | buffer identifier |
| *present* | output | boolean | presence indicator |
| *buffer_OUT* | output | APEX_Buffer_type | buffer |

  Processing:
  When there is a buffer identified by *buffer_ID*, the presence indicator is `TRUE` and *buffer_OUT* is the corresponding buffer. Otherwise, *present* is `FALSE`.

```
process BUFFER_CHECKID =
      ( ? Comm_ComponentID_type buffer_ID;
        ! boolean present;
          APEX_Buffer_type buffer_OUT;
      )
      spec (| (| buffer_ID -> present
               | { buffer_ID -> buffer_OUT } when present
               |)
            | (| buffer_ID ^= present
               | buffer_OUT ^= when present
               |)
            |)
 ;
```

- process BUFFER_CHECKNAME
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *buffer_name* | input | Comm_ComponentName_type | buffer name |
| *present* | output | boolean | presence indicator |
| *buffer_ID* | output | Comm_ComponentID_type | buffer identifier |

  Processing:
  When there is a buffer named *buffer_name*, the presence indicator is `TRUE` and *buffer_ID* is the corresponding identifier. Otherwise *present* is `FALSE`.

```
process BUFFER_CHECKNAME =
     ( ? Comm_ComponentName_type buffer_name;
       ! boolean present;
         Comm_ComponentID_type buffer_ID;
     )
     spec (| (| buffer_name -> present
              | { buffer_name -> buffer_ID } when present
              |)
           | (| buffer_name ^= present
              | buffer_ID ^= when present
              |)
           |)
;
```

- process BUFFER_RECORD
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *buffer_name* | input | Comm_ComponentName_type | buffer name |
| *buffer_size* | input | integer | buffer size limit |
| *message_size* | input | integer | message size |
| *queuing_discipline* | input | QueuingDiscipline_type | queuing discipline |
| *buffer_ID* | output | Comm_ComponentID_type | blackboard identifier |
| *recorded* | output | boolean | record indicator |

  Processing:
  This service records the buffer named *buffer_name*. Its size limit is *buffer_size*. The size of the messages stored in this buffer is *message_size*. The queuing discipline of the process waiting queue associated with the buffer is specified by the input *queuing_discipline*. The buffer identifier is given by the output *buffer_ID*. The record indicator carries the value `TRUE` when the buffer is actually recorded; otherwise, it is `FALSE`.

```
process BUFFER_RECORD =
     ( ? Comm_ComponentName_type buffer_name;
         BufferSize_type buffer_size;
         MessageSize_type message_size;
         QueuingDiscipline_type queuing_discipline;
       ! Comm_ComponentID_type buffer_ID;
         boolean recorded;
     )
     spec (| (| { buffer_name,buffer_size,message_size,
                  queuing_discipline } -> recorded
              | { { buffer_name,buffer_size,message_size,
                  queuing_discipline } -> buffer_ID } when recorded
              |)
           | (| buffer_name ^= buffer_size ^= message_size ^=
                  queuing_discipline ^= recorded
              | buffer_ID ^= when recorded
              |)
           |)
;
```

- process BUFFER_UPDATE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *buffer_IN* | input | APEX_Buffer_type | buffer |

  Processing:
  This service updates the buffer named *buffer_IN*.Name in the manager with the status of the buffer *buffer_IN*.

```
process BUFFER_UPDATE =
```

```
        ( ? APEX_Buffer_type buffer_IN;
        )
    ;
```

- process EVENT_CHECKID
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *event_ID* | input | Comm_ComponentID_type | event identifier |
| *present* | output | boolean | presence indicator |
| *event_OUT* | output | APEX_Event_type | event |

  Processing:
  When there is an event identified by *event_ID*, the presence indicator is `TRUE` and *event_OUT* is the corresponding event. Otherwise, *present* is `FALSE`.

```
process EVENT_CHECKID =
     ( ? Comm_ComponentID_type event_ID;
       ! boolean present;
         APEX_Event_type event_OUT;
     )
     spec (| (| event_ID -> present
              | { event_ID -> event_OUT } when present
              |)
            | (| event_ID ^= present
               | event_OUT ^= when present
               |)
          |)
;
```

- process EVENT_CHECKNAME
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *event_name* | input | Comm_ComponentName_type | event name |
| *present* | output | boolean | presence indicator |
| *event_ID* | output | Comm_ComponentID_type | event identifier |

  Processing:
  When there is an event named *event_name*, the presence indicator is `TRUE` and *event_ID* is the corresponding identifier. Otherwise, *present* is `FALSE`.

```
process EVENT_CHECKNAME =
     ( ? Comm_ComponentName_type event_name;
       ! boolean present;
         Comm_ComponentID_type event_ID;
     )
     spec (| (| event_name -> present
```

```
                | { event_name -> event_ID } when present
                |)
            | (| event_name ^= present
                | event_ID ^= when present
                |)
            |)
;
```

- process EVENT_RECORD
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *event_name* | input | Comm_ComponentName_type | event name |
| *event_ID* | output | Comm_ComponentID_type | event identifier |
| *recorded* | output | boolean | record indicator |

  Processing:
  This service records the event named *event_name*. The associated identifier is *event_ID*. The record indicator carries the value `TRUE` when the event is actually recorded; otherwise, it is `FALSE`.

```
process EVENT_RECORD =
     ( ? Comm_ComponentName_type event_name;
       ! Comm_ComponentID_type event_ID;
         boolean recorded;
     )
     spec (| (| event_name -> recorded
              | { event_name -> event_ID } when recorded
              |)
           | (| event_name ^= recorded
              | event_ID ^= when recorded
              |)
           |)
;
```

- process EVENT_UPDATE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *event_IN* | input | APEX_Event_type | event |

  Processing:
  This service updates the event named *event_IN*.Name in the manager with the status of the event *event_IN*.

```
process EVENT_UPDATE =
     ( ? APEX_Event_type event_IN;
     )
;
```

- process SEMAPHORE_CHECKID
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *semaphore_ID* | input | Comm_ComponentID_type | semaphore identifier |
| *present* | output | boolean | presence indicator |
| *semaphore_OUT* | output | APEX_Semaphore_type | semaphore |

  Processing:
  When there is a semaphore identified by *semaphore_ID*, the presence indicator is `TRUE` and *semaphore_OUT* is the corresponding semaphore. Otherwise, *present* is `FALSE`.

```
process SEMAPHORE_CHECKID =
     ( ? Comm_ComponentID_type semaphore_ID;
       ! boolean present;
         APEX_Semaphore_type semaphore_OUT;
     )
     spec (| (| semaphore_ID -> present
              | { semaphore_ID -> semaphore_OUT } when present
              |)
           | (| semaphore_ID ^= present
              | semaphore_OUT ^= when present
              |)
           |)
;
```

- process SEMAPHORE_CHECKNAME
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *semaphore_name* | input | Comm_ComponentName_type | semaphore name |
| *present* | output | boolean | presence indicator |
| *semaphore_ID* | output | Comm_ComponentID_type | semaphore identifier |

  Processing:
  When there is a semaphore named *semaphore_name*, the presence indicator is `TRUE` and *semaphore_ID* is the corresponding identifier. Otherwise, *present* is `FALSE`.

```
process SEMAPHORE_CHECKNAME =
     ( ? Comm_ComponentName_type semaphore_name;
       ! boolean present;
         Comm_ComponentID_type semaphore_ID;
     )
     spec (| (| semaphore_name -> present
              | { semaphore_name -> semaphore_ID } when present
              |)
           | (| semaphore_name ^= present
              | semaphore_ID ^= when present
              |)
           |)
;
```

- process SEMAPHORE_RECORD
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *semaphore_name* | input | Comm_ComponentName_type | semaphore name |
| *current_value* | input | integer | current value |
| *maximum_value* | input | integer | maximum value |
| *queuing_discipline* | input | QueuingDiscipline_type | queuing discipline |
| *semaphore_ID* | output | Comm_ComponentID_type | semaphore identifier |
| *recorded* | output | boolean | record indicator |

  Processing:
  This service records a semaphore named *semaphore_name*. Its initial and maximum values are respectively *current_value* and *maximum_value*. The queuing discipline of the process waiting queue associated with the semaphore is specified by the input *queuing_discipline*. The record indicator carries the value `TRUE` when the semaphore is actually recorded; otherwise, it is `FALSE`.

```
process SEMAPHORE_RECORD =
     ( ? Comm_ComponentName_type semaphore_name;
         SemaphoreValue_type current_value;
         SemaphoreValue_type maximum_value;
         QueuingDiscipline_type queuing_discipline;
       ! Comm_ComponentID_type semaphore_ID;
         boolean recorded;
     )
     spec (| (| { semaphore_name,current_value,maximum_value,
                  queuing_discipline } -> recorded
              | { { semaphore_name,current_value,maximum_value,
                  queuing_discipline } -> semaphore_ID } when recorded
              |)
           | (| semaphore_name ^= recorded ^= current_value ^=
                  maximum_value ^= queuing_discipline
              | semaphore_ID ^= when recorded
              |)
           |)
;
```

- process SEMAPHORE_UPDATE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *semaphore_IN* | input | APEX_Semaphore_type | semaphore |

  Processing:
  This service updates the semaphore named *semaphore_IN*.Name in the manager with the status of the semaphore *semaphore_IN*.

```
process SEMAPHORE_UPDATE =
```

```
        ( ? APEX_Semaphore_type semaphore_IN;
        )
    ;
```

- process BLACKBOARD_CHECKCAPACITY
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *full* | output | boolean | board manager status |

  Processing:
  This service checks whether or not the blackboard manager is full.

```
process BLACKBOARD_CHECKCAPACITY =
     ( ! boolean full;
     )
;
```

- process BUFFER_CHECKCAPACITY
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *full* | output | boolean | buffer manager status |

  Processing:
  This service checks whether or not the buffer manager is full.

```
process BUFFER_CHECKCAPACITY =
     ( ! boolean full;
     )
;
```

- process EVENT_CHECKCAPACITY
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *full* | output | boolean | event manager status |

  Processing:
  This service checks whether or not the event manager is full.

```
process EVENT_CHECKCAPACITY =
     ( ! boolean full;
     )
;
```

- process SEMAPHORE_CHECKCAPACITY
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *full* | output | boolean | sema manager status |

  Processing:
  This service checks whether or not the semaphore manager is full.

```
process SEMAPHORE_CHECKCAPACITY =
     ( ! boolean full;
     )
;
```

### 8.3.3 Communication and synchronization services

For each mechanism (blackboard, buffer, event and semaphore), we present the associated services.

1. BLACKBOARD services

   - process CREATE_BLACKBOARD*
     Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *blackboard_name* | input | Comm_ComponentName_type | blackboard name |
| *message_size* | input | MessageSize_type | message size |
| *blackboard_ID* | output | Comm_ComponentID_type | blackboard ident |
| *return_code* | output | ReturnCode_type | diagnostic |

     Processing:
     This service is used to create a blackboard named *blackboard_name*. If the request satisfies all the creation conditions, a blackboard identifier (*blackboard_ID*) is returned. The output *return_code* indicates the diagnostic of the request.

     (a) The return code value is:
     - `INVALID_CONFIG` when there is not enough space for creating a new blackboard;
     - `NO_ACTION` when a blackboard with the same name has been already created;
     - `INVALID_PARAM` when the input *message_size* is out of range;
     - `INVALID_MODE` when the operating mode is `NORMAL`;
     - `NO_ERROR` otherwise.

     (b) In the case of successful completion, *blackboard_ID* denotes the identifier of an unallocated blackboard control block. The empty indicator is set to `TRUE`.

```
process CREATE_BLACKBOARD =
     ( ? Comm_ComponentName_type blackboard_name;
         MessageSize_type message_size;
       ! Comm_ComponentID_type blackboard_ID;
```

```
        ReturnCode_type return_code;
    )
    spec (| (| { blackboard_name, message_size } -> return_code
             | { { blackboard_name, message_size } -> blackboard_ID }
                     when (return_code = #NO_ERROR)
             |)
          | (| return_code ^= message_size ^= blackboard_name
             | blackboard_ID ^= when (return_code = #NO_ERROR)
             |)
          |)
;
```

- process DISPLAY_BLACKBOARD*
  Interface:

| **Label** | **Nature** | **Type** | **Comments** |
|---|---|---|---|
| *blackboard_ID* | input | Comm_ComponentID_type | blackboard ident |
| *message* | input | MessageArea_type | message address |
| *length* | input | MessageSize_type | message size |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to display a message (defined by inputs *message* and *length*) in the blackboard specified by *blackboard_ID*.

  (a) The return code value is:
  - `INVALID_PARAM` when the input *blackboard_ID* does not identify any blackboard, or the message is too long;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion, the empty indicator is set to `FALSE`. The contents of the specified blackboard is overwritten. If there are processes blocked on an empty blackboard, for each of them, the state is set to `READY` (except if another process suspended it). If some of the blocked processes are waiting with a time-out, the affected time counters are stopped. Finally, process scheduling is performed if preemption is enabled.

```
process DISPLAY_BLACKBOARD =
    ( ? Comm_ComponentID_type blackboard_ID;
        MessageArea_type message;
        MessageSize_type length;
      ! ReturnCode_type return_code;
    )
    spec (| (| { blackboard_ID, message, length }
                  -> return_code |)
          | (| return_code ^= blackboard_ID ^= message
                  ^= length |)
          |)
;
```

- process READ_BLACKBOARD*

Interface

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | parameter | ProcessID_type | caller ident |
| *blackboard_ID* | input | Comm_ComponentID_type | blackboard ident |
| *timeout* | input | SystemTime_type | waiting duration |
| *message* | output | MessageArea_type | message address |
| *length* | output | MessageSize_type | message size |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service is used to read a message (defined by outputs *message* and *length*) from the blackboard specified by *blackboard_ID*. If the blackboard is empty, the calling process (*process_ID*) goes into a waiting state.

(a) The return code value is:
- `INVALID_PARAM` when the input *blackboard_ID* does not identify any blackboard, or *timeout* is out of range;
- `NOT_AVAILABLE` when the input *timeout* value is zero;
- `INVALID_MODE` when preemption is disabled, or the parameter *process_ID* identifies an error handler process;
- `NO_ERROR` otherwise.

(b) In the case of successful completion,
- if the blackboard is not empty, the output message is the one currently displayed in the blackboard.
- Otherwise, the calling process goes into waiting state. A time counter with duration *timeout* is initiated if *timeout* is not infinite. Process scheduling is performed if preemption is enabled. Finally, the last available message of the blackboard is sent.

```
process READ_BLACKBOARD =
     { ProcessID_type process_ID; }
     ( ? Comm_ComponentID_type blackboard_ID;
         SystemTime_type timeout;
       ! MessageArea_type message;
         MessageSize_type length;
         ReturnCode_type return_code;
     )
     spec (| (| { { blackboard_ID, timeout } ->
                    return_code } when C_return_code
              | { { blackboard_ID, timeout } ->
                    { message, length } }
                    when (return_code = #NO_ERROR)
              |)
           | (| blackboard_ID ^= timeout ^= C_return_code
              | return_code ^= when C_return_code
              | message ^= length ^= when (return_code = #NO_ERROR)
              |)
           |)
```

```
        where boolean C_return_code;
    end;
```

- process CLEAR_BLACKBOARD*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *blackboard_ID* | input | Comm_ComponentID_type | blackboard ident |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to clear the blackboard specified by *blackboard_ID*.
  (a) The return code value is:
    - `INVALID_PARAM` when the input *blackboard_ID* does not identify any blackboard in the current partition;
    - `NO_ERROR` otherwise.
  (b) In the case of successful completion, the empty indicator is set to `TRUE`.

```
process CLEAR_BLACKBOARD =
     ( ? Comm_ComponentID_type blackboard_ID;
       ! ReturnCode_type return_code;
     )
     spec (| (| blackboard_ID -> return_code |)
           | (| blackboard_ID ^= return_code |)
           |)
;
```

- process GET_BLACKBOARD_ID*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *blackboard_name* | input | Comm_ComponentName_type | blackboard name |
| *blackboard_ID* | output | Comm_ComponentID_type | blackboard ident |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service allows to get the identifier (*blackboard_ID*) of the blackboard named *blackboard_name*.
  (a) The return code value is:
    - `INVALID_CONFIG` when no blackboard is named *blackboard_name* in the current partition;
    - `NO_ERROR` otherwise.
  (b) In the case of successful completion, *blackboard_ID* denotes the identifier of the blackboard named *blackboard_name*.

```
process GET_BLACKBOARD_ID =
     ( ? Comm_ComponentName_type blackboard_name;
```

```
        ! Comm_ComponentID_type blackboard_ID;
          ReturnCode_type return_code;
     )
     spec (| (| blackboard_name -> return_code
              | { blackboard_name -> blackboard_ID }
                  when (return_code = #NO_ERROR)
              |)
           | (| blackboard_name ^= return_code
              | blackboard_ID ^= when (return_code = #NO_ERROR)
              |)
           |)
;
```

- process GET_BLACKBOARD_STATUS*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *blackboard_ID* | input | Comm_ComponentID_type | blackboard ident |
| *blackboard_status* | output | BlackboardStatus_type | its current status |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service allows to get the status (*blackboard_status*) of the blackboard identified by *blackboard_ID*.

  (a) The return code value is:
  - `INVALID_PARAM` when no blackboard is identified by *blackboard_ID* in the current partition;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion, *blackboard_status* denotes the current status of the blackboard identified by *blackboard_ID*.

```
process GET_BLACKBOARD_STATUS =
     ( ? Comm_ComponentID_type blackboard_ID;
       ! BlackboardStatus_type blackboard_status;
         ReturnCode_type return_code;
     )
     spec (| (| blackboard_ID -> return_code
              | { blackboard_ID -> blackboard_status }
                  when (return_code = #NO_ERROR)
              |)
           | (| blackboard_ID ^= return_code
              | blackboard_status ^= when (return_code = #NO_ERROR)
              |)
           |)
;
```

2. BUFFER services

- process CREATE_BUFFER*
  Interface:

| **Label** | **Nature** | **Type** | **Comments** |
|---|---|---|---|
| *buffer_name* | input | Comm_ComponentName_type | buffer name |
| *buffer_size* | input | BufferSize_type | size limit |
| *message_size* | input | MessageSize_type | size of messages |
| *queuing_discipline* | input | QueuingDiscipline_type | queuing rule |
| *buffer_ID* | output | Comm_ComponentID_type | buffer identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to create a message buffer (named *buffer_name*). If the request satisfies all the creation conditions, a buffer identifier (*buffer_ID*) is returned. The size limit of this buffer is *buffer_size*. The input *queuing_discipline* specifies the queuing discipline of waiting processes on the buffer.

  (a) The return code value is:
  - `INVALID_CONFIG` when there is not enough place for creating a new buffer;
  - `NO_ACTION` when a buffer with the same name has been already created;
  - `INVALID_PARAM` when at least one of the inputs *buffer_size* and *message_size* is out of range, or *queuing_discipline* is invalid;
  - `INVALID_MODE` when the operating mode is `NORMAL`;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion, *buffer_ID* denotes an identifier of an unallocated buffer control block. The process buffer queuing rule is set to *queuing_discipline*.

```
process CREATE_BUFFER =
     ( ? Comm_ComponentName_type buffer_name;
         BufferSize_type buffer_size;
         MessageSize_type message_size;
         QueuingDiscipline_type queuing_discipline;
       ! Comm_ComponentID_type buffer_ID;
         ReturnCode_type return_code;
     )
     spec (| (| { buffer_size, message_size, queuing_discipline,
                  buffer_name } -> return_code
              | { { buffer_size, message_size, queuing_discipline,
                  buffer_name } -> buffer_ID }
                  when (return_code = #NO_ERROR)

              |)
            | (| return_code ^= buffer_size ^= message_size
                   ^= queuing_discipline ^= buffer_name
              | buffer_ID ^= when (return_code = #NO_ERROR)
              |)
            |)
;
```

- process SEND_BUFFER*

Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | parameter | ProcessID_type | caller identifier |
| *buffer_ID* | input | Comm_ComponentID_type | buffer identifier |
| *message* | input | MessageArea_type | message address |
| *length* | input | MessageSize_type | message size |
| *timeout* | input | SystemTime_type | waiting duration |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service is used to send a message in a buffer identified by *buffer_ID*.

(a) The return code value is:
- `INVALID_PARAM` when the input *buffer_ID* does not identify any buffer, or message is too long, or *timeout* is out of range;
- `NOT_AVAILABLE` when the input *timeout* value is zero;
- `INVALID_MODE` when preemption is disabled, or the parameter *process_ID* identifies an error handler process;
- `NO_ERROR` otherwise.

(b) In the case of successful completion,
- when the buffer is not full,
  - if no process is waiting on an empty buffer, the message is stored in the buffer.
  - else the first process is removed from the process queue. It retrieves the current message. If this process is waiting with a time-out, the affected time counter is stopped. Its state becomes `READY` (except if another process suspended it).
- Otherwise, the calling process goes into waiting state. It is inserted in the process waiting queue associated with the buffer, at the position specified by the queuing rule. A time counter with duration *timeout* is initiated if *timeout* is not infinite.

Finally, process scheduling is performed if preemption is enabled.

```
process SEND_BUFFER =
     { ProcessID_type process_ID; }
     ( ? Comm_ComponentID_type buffer_ID;
         MessageArea_type message;
         MessageSize_type length;
         SystemTime_type timeout;
       ! ReturnCode_type return_code;
     )
     spec (| (| { { buffer_ID, timeout, message, length }
                      -> return_code } when C_return_code |)
           | (| buffer_ID ^= timeout ^= message
                  ^= length ^= C_return_code
              | return_code ^= when C_return_code
```

```
            |)
          |)
    where boolean C_return_code;
end;
```

- process RECEIVE_BUFFER*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | parameter | ProcessID_type | caller identifier |
| *buffer_ID* | input | Comm_ComponentID_type | buffer identifier |
| *timeout* | input | SystemTime_type | waiting maximum |
| *message* | output | MessageArea_type | message address |
| *length* | output | MessageSize_type | message size |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to receive a message from the buffer identified by *buffer_ID*.

  (a) The return code value is:
  - `INVALID_PARAM` when the input *buffer_ID* does not identify any buffer, or *timeout* is out of range;
  - `NOT_AVAILABLE` when the input *timeout* value is zero;
  - `INVALID_MODE` when preemption is disabled, or the parameter *process_ID* identifies an error handler process;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion,
  - when the buffer is not empty, the message is the first message of the specified buffer message queue. If there are processes waiting on a full buffer, the first process is removed from the process queue. The message sent by this process is put in the message queue. If the process is waiting with a time-out, the affected time counter is stopped. Its state becomes `READY` (except if another process suspended it).
  - Otherwise, the calling process goes into a waiting state. It is inserted in the process waiting queue associated with the buffer, at the position specified by the queuing rule parameter. A time counter with duration *timeout* is initiated if *timeout* is not infinite.

  Finally, process scheduling is performed if preemption is enabled.

```
process RECEIVE_BUFFER =
     { ProcessID_type process_ID; }
     ( ? Comm_ComponentID_type buffer_ID;
         SystemTime_type timeout;
       ! MessageArea_type message;
         MessageSize_type length;
         ReturnCode_type return_code;
     )
```

```
        spec (| (| { { buffer_ID, timeout } -> return_code }
                        when C_return_code
                 | { { buffer_ID, timeout } -> { message, length }
                        when (return_code = #NO_ERROR)
                 |)
               | (| buffer_ID ^= timeout ^= C_return_code
                 | return_code ^= when C_return_code
                 | message ^= length ^= when (return_code = #NO_ERROR)
                 |)
               |)
        where boolean C_return_code;
    end;
```

- process GET_BUFFER_ID*
  Interface:

| **Label** | **Nature** | **Type** | **Comments** |
|---|---|---|---|
| *buffer_name* | input | Comm_ComponentName_type | buffer name |
| *buffer_ID* | output | Comm_ComponentID_type | buffer identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service allows to get the identifier (*buffer_ID*) of the buffer named *buffer_name*.

  (a) The return code value is:
  - `INVALID_CONFIG` when no buffer is named *buffer_name* in the current partition;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion, *buffer_ID* denotes the identifier of the buffer named *buffer_name*.

```
process GET_BUFFER_ID =
     ( ? Comm_ComponentName_type buffer_name;
       ! Comm_ComponentID_type buffer_ID;
         ReturnCode_type return_code;
     )
     spec (| (| buffer_name -> return_code
              | { buffer_name -> buffer_ID }
                   when (return_code = #NO_ERROR)
              |)
            | (| buffer_name ^= return_code
              | buffer_ID ^= when (return_code = #NO_ERROR)
              |)
            |)
;
```

- process GET_BUFFER_STATUS*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *buffer_ID* | input | Comm_ComponentID_type | buffer identifier |
| *buffer_status* | output | BufferStatus_type | current status |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service allows to get the status (*buffer_status*) of the buffer identified by *buffer_ID*.

  (a) The return code value is:
  - `INVALID_PARAM` when no buffer is identified by *buffer_ID* in the current partition;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion, *buffer_status* denotes the current status of the buffer identified by *buffer_ID*.

```
process GET_BUFFER_STATUS =
     ( ? Comm_ComponentID_type buffer_ID;
       ! BufferStatus_type buffer_status;
         ReturnCode_type return_code;
     )
     spec (| (| buffer_ID -> return_code
              | { buffer_ID -> buffer_status }
                   when (return_code = #NO_ERROR)
              |)
           | (| return_code ^= buffer_ID
              | buffer_status ^= when (return_code = #NO_ERROR)
              |)
           |)
;
```

3. EVENT services

- process CREATE_EVENT*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *event_name* | input | Comm_ComponentName_type | event name |
| *event_ID* | output | ComponentID_type | event identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to create an event called *event_name*. If the request satisfies all the creation conditions, an event identifier (*event_ID*) is returned.

  (a) The return code value is:
  - `INVALID_CONFIG` when there is not enough place for creating a new event;
  - `NO_ACTION` when an event with the same name had already been created;
  - `INVALID_MODE` when the operating mode is `NORMAL`;

- NO_ERROR otherwise.

(b) In the case of successful completion, *event_ID* denotes an identifier of unallocated event. The event state is set to DOWN.

```
process CREATE_EVENT =
     ( ? Comm_ComponentName_type event_name;
       ! Comm_ComponentID_type event_ID;
         ReturnCode_type return_code;
     )
     spec (| (| event_name -> return_code
              | { event_name -> event_ID } when (return_code = #NO_ERROR)
              |)
           | (| event_name ^= return_code
              | event_ID ^= when (return_code = #NO_ERROR)
              |)
           |)
;
```

- process SET_EVENT*

Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *event_ID* | input | Comm_ComponentID_type | event identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service is used to set the specified event state to "up". A

(a) The return code value is:

- INVALID_PARAM when the input *event_ID* does not identify any event;
- NO_ERROR otherwise.

(b) In the case of successful completion, the event state is set to UP. If there are processes waiting for that event, their state is set to READY (except if another process suspended them previously). If some of the blocked processes are waiting with a time-out, the affected time counters are stopped. Finally, process scheduling is performed if preemption is enabled.

```
process SET_EVENT =
     ( ? Comm_ComponentID_type event_ID;
       ! ReturnCode_type return_code;
     )
     spec (| (| event_ID -> return_code |)
           | (| event_ID ^= return_code |)
           |)
;
```

- process RESET_EVENT*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *event_ID* | input | Comm_ComponentID_type | event identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to set the specified event state to "down".

  (a) The return code value is:
  - `INVALID_PARAM` when the input *event_ID* does not identify any event;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion, the event state is set to `DOWN`.

```
process RESET_EVENT =
     ( ? Comm_ComponentID_type event_ID;
       ! ReturnCode_type return_code;
     )
     spec (| (| event_ID -> return_code |)
           | (| event_ID ^= return_code |)
           |)
;
```

- process WAIT_EVENT*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | parameter | ProcessID_type | caller identifier |
| *event_ID* | input | Comm_ComponentID_type | event identifier |
| *timeout* | input | SystemTime_type | waiting duration |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to move the calling process from running state to waiting state if the specified event state is “down” and if the specified time-out is not zero. It goes on executing if the state is “up”.

  (a) The return code value is:
  - `INVALID_PARAM` when the input *event_ID* does not identify any event, or *timeout* is out of range;
  - `NOT_AVAILABLE` when the input *timeout* value is zero;
  - `INVALID_MODE` when preemption is disabled, or the parameter *process_ID* identifies an error handler process;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion, if the state of the specified event is “down”, the calling process goes into waiting state. A time counter with duration *timeout* is initiated if *timeout* is not infinite. Finally, process scheduling is performed if preemption is enabled.

```
process WAIT_EVENT =
     { ProcessID_type process_ID; }
     ( ? Comm_ComponentID_type event_ID;
         SystemTime_type timeout;
       ! ReturnCode_type return_code;
     )
     spec (| (| { { event_ID, timeout } -> when C_return_code |)
           | (| event_ID ^= timeout ^= C_return_code
              | return_code ^= when C_return_code
              |)
           |)
     where boolean C_return_code;
end;
```

- process GET_EVENT_ID*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *event_name* | input | Comm_ComponentName_type | event name |
| *event_ID* | output | Comm_ComponentID_type | event identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service allows to get the identifier (*event_ID*) of the event named *event_name*.

  (a) The return code value is:
  - `INVALID_CONFIG` when no event is named *event_name* in the current partition;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion, *event_ID* denotes the identifier of the event named *event_name*.

```
process GET_EVENT_ID =
     ( ? Comm_ComponentName_type event_name;
       ! Comm_ComponentID_type event_ID;
         ReturnCode_type return_code;
     )
     spec (| (| event_name -> return_code
              | { event_name -> event_ID }
                   when (return_code = #NO_ERROR)
              |)
           | (| event_name ^= return_code
              | event_ID ^= when (return_code = #NO_ERROR)
              |)
           |)
;
```

- process GET_EVENT_STATUS*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *event_ ID* | input | Comm_ComponentID_type | event identifier |
| *event_ status* | output | EventStatus_type | current status |
| *return_ code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service allows to get the status (*event_ status*) of the event identified by *event_ ID*.
  (a) The return code value is:
    - `INVALID_PARAM` when no event is identified by *event_ ID* in the current partition;
    - `NO_ERROR` otherwise.
  (b) In the case of successful completion, *event_ status* denotes the current status of the event identified by *event_ ID*.

```
process GET_EVENT_STATUS =
     ( ? Comm_ComponentID_type event_ID;
       ! EventStatus_type event_status;
         ReturnCode_type return_code;
     )
     spec (| (| event_ID -> return_code
              | { event_ID -> event_status }
                   when (return_code = #NO_ERROR)
              |)
           | (| event_ID ^= return_code
              | event_status ^= when (return_code = #NO_ERROR)
              |)
           |)
;
```

4. SEMAPHORE services

- process CREATE_SEMAPHORE*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *semaphore_ name* | input | Comm_ComponentName_type | semaphore name |
| *current_ value* | input | integer | its current value |
| *maximum_ value* | input | integer | its max value |
| *queuing_ discipline* | input | QueuingDiscipline_type | queuing rule |
| *semaphore_ ID* | output | Comm_ComponentID_type | semaphore ident |
| *return_ code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to create a semaphore named *semaphore_ name*. If the request satisfies all the creation conditions, a semaphore identifier (*semaphore_ ID*) is returned. The input *queuing_ discipline* specifies the queuing discipline of waiting processes on the semaphore.

(a) The return code value is:
- INVALID_CONFIG when there is not enough place for creating a new semaphore;
- NO_ACTION when a semaphore with the same name has been already created;
- INVALID_PARAM when at least one of the inputs *current_value* and *maximum_value* is out of range, or *queuing_discipline* is invalid;
- INVALID_MODE when the operating mode is NORMAL;
- NO_ERROR otherwise.

(b) In the case of successful completion, *semaphore_ID* denotes an identifier of unallocated semaphore control block. The process semaphore queuing discipline is set to *queuing_discipline*. Furthermore, the semaphore is initialized with *current_value* and *maximum_value*.

```
process CREATE_SEMAPHORE =
     ( ? Comm_ComponentName_type semaphore_name;
         SemaphoreValue_type current_value;
         SemaphoreValue_type maximum_value;
         QueuingDiscipline_type queuing_discipline;
       ! Comm_ComponentID_type semaphore_ID;
         ReturnCode_type return_code;
     )
     spec (| (| { semaphore_name, current_value, maximum_value,
                  queuing_discipline } -> return_code
             | { { semaphore_name, current_value, maximum_value,
                   queuing_discipline } -> semaphore_ID }
                   when (return_code = #NO_ERROR)
             |)
           | (| return_code ^= semaphore_name ^= current_value
                ^= maximum_value ^= queuing_discipline
             | semaphore_ID ^= when (return_code = #NO_ERROR)
             |)
           |)
;
```

- process WAIT_SEMAPHORE*

Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | parameter | ProcessID_type | caller identifier |
| *semaphore_ID* | input | Comm_ComponentID_type | semaphore ident |
| *timeout* | input | SystemTime_type | waiting duration |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service is used to move the calling process from running state to waiting state if the specified semaphore value is zero and if the specified time-out is not zero. It goes on executing if the value is greater than zero.

(a) The return code value is:

- `INVALID_PARAM` when the input *event_ID* does not identify any event, or *timeout* is out of range;
- `NOT_AVAILABLE` when the input *timeout* value is zero;
- `INVALID_MODE` when preemption is disabled, or the parameter *process_ID* identifies an error handler process;
- `NO_ERROR` otherwise.

(b) In the case of successful completion,

- if the current value is greater than zero, it is decremented by one;
- otherwise, the calling process goes into waiting state. It is inserted in the semaphore process queue at the position specified by the queuing rule. A time counter with duration *timeout* is initiated if *timeout* is not infinite. Finally, process scheduling is performed if preemption is enabled.

```
process WAIT_SEMAPHORE =
     { ProcessID_type process_ID; }
     ( ? Comm_ComponentID_type semaphore_ID;
         SystemTime_type timeout;
       ! ReturnCode_type return_code;
     )
     spec (| (| { { semaphore_ID, timeout }
                  -> return_code } when C_return_code
              |)
           | (| semaphore_ID ^= timeout ^= C_return_code
              | return_code ^= when C_return_code
              |)
           |)
     where boolean C_return_code;
end;
```

- process SIGNAL_SEMAPHORE*

Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *semaphore_ID* | input | Comm_ComponentID_type | semaphore ident |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:

This service is used to increment the current value of the specified semaphore.

(a) The return code value is:

- `INVALID_PARAM` when the input *semaphore_ID* does not identify any blackboard;
- `NO_ACTION` when the maximum value has been already reached;
- `NO_ERROR` otherwise.

(b) In the case of successful completion, the current value of the specified semaphore is incremented. If there are processes waiting on that semaphore, the first is removed from the semaphore queue. Its state is set to `READY` (except if another process suspended itpreviously). If it is waiting with a time-out, the affected

time counter is stopped. Finally, process scheduling is performed if preemption is enabled.

```
process SIGNAL_SEMAPHORE =
     ( ? Comm_ComponentID_type semaphore_ID;
       ! ReturnCode_type return_code;
     )
     spec (| (| semaphore_ID -> return_code |)
           | (| semaphore_ID ^= return_code |)
           |)
;
```

- process GET_SEMAPHORE_STATUS*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *semaphore_ID* | input | Comm_ComponentID_type | semaphore ident |
| *semaphore_status* | output | SemaphoreStatus_type | current status |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service allows to get the status (*semaphore_status*) of the semaphore identified by *semaphore_ID*.

  (a) The return code value is:
  - `INVALID_PARAM` when no semaphore is identified by *semaphore_ID* in the current partition;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion, *semaphore_status* denotes the current status of the semaphore identified by *semaphore_ID*.

```
process GET_SEMAPHORE_STATUS =
     ( ? Comm_ComponentID_type semaphore_ID;
       ! SemaphoreStatus_type semaphore_status;
         ReturnCode_type return_code;
     )
     spec (| (| semaphore_ID -> return_code
              | { semaphore_ID -> semaphore_status }
                   when (return_code = #NO_ERROR)
              |)
           | (| semaphore_ID ^= return_code
              | semaphore_status ^= when (return_code = #NO_ERROR)
              |)
           |)
;
```

- process GET_SEMAPHORE_ID*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *semaphore_name* | input | Comm_ComponentName_type | semaphore name |
| *semaphore_ID* | output | Comm_ComponentID_type | semaphore ident |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service allows to get the identifier (*semaphore_ID*) of the semaphore named *semaphore_name*.
  (a) The return code value is:
    - `INVALID_CONFIG` when no semaphore is named *semaphore_name* in the current partition;
    - `NO_ERROR` otherwise.
  (b) In the case of successful completion, *semaphore_ID* denotes the identifier of the semaphore named *semaphore_name*.

```
process GET_SEMAPHORE_ID =
     ( ? Comm_ComponentName_type semaphore_name;
       ! Comm_ComponentID_type semaphore_ID;
         ReturnCode_type return_code;
     )
     spec (| (| semaphore_name -> return_code
              | { semaphore_name -> semaphore_ID }
                   when (return_code = #NO_ERROR)
              |)
           | (| semaphore_name ^= return_code
              | semaphore_ID ^= when (return_code = #NO_ERROR)
              |)
           |)
;
```

## 8.4 Inter-partition communication mechanisms

We first remind the main intra-partition communication and synchronization mechanisms. Then, we give the associated services.

1. **Sampling port** is a communication object used by partitions. Each new occurrence of a message overwrites the previous. Messages have a fixed length. A refresh period attribute applies to ports. A validity output parameter indicates whether the age of the read message is consistent with the required refresh period attribute of the port.
2. **Queuing port**: it is also a communication object. Messages are stored in FIFO order, and they have a variable length.

### 8.4.1 Types

Here also, we give information about the types used by the services.

| Types | Nature |
|---|---|
| PortDirection_type | (SOURCE, DESTINATION); |
| SamplingPortSize_type | integer; |

| Label | Nature |
|---|---|
| SamplingPortStatus_type | struct ( SamplingPortSize_type SamplingPort_Size;<br>PortDirection_type Port_Direction;<br>SystemTime_type Refresh_Period;<br>boolean Validity;) |
| APEX_SamplingPort_type | struct ( Comm_ComponentName_type SamplingPort_Name;<br>SamplingPortSize_type Sampling<br>-Port_Size;<br>PortDirection_type Port_Direction;<br>SystemTime_type Refresh_Period;<br>boolean Validity;<br>boolean Empty;<br>SystemTime_type Message_Age;<br>Message_type Message;) |
| QueuingPortStatus_type | struct ( QueuingPortSize_type QueuingPort_Size;<br>PortDirection_type Port_Direction;<br>MessageRange_type Nb_Message;<br>WaitingRange_type Waiting_Processes;) |
| APEX_QueuingPort_type | struct ( Comm_ComponentName_type QueuingPort_Name;<br>QueuingPortSize_type QueuingPort_Size;<br>PortDirection_type Port_Direction;<br>MessageSize_type Message_Size;<br>MessageRange_type Nb_Message;<br>[MAX_QUEUING_PORT_SIZE] Message_type<br>Message_Range;<br>QueuingDiscipline_type Queuing_Discipline;<br>WaitingRange_type WaitingOnSend_Processes;<br>WaitingRange_type WaitingOnReceive_Processes;<br>QueueAddress_type WaitingOnSend_Queue;<br>QueueAddress_type WaitingOnReceive_Queue;) |

### 8.4.2 Inter-partition communication mechanisms manager

Here also, the mechanisms are uniquely identified in their corresponding managers. The management services are given bellow.

- process QUEUINGPORT_CHECKID
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *queuingPort_ID* | input | Comm_ComponentID_type | queuing port ident. |
| *present* | output | boolean | presence indicator |
| *queuingPort_OUT* | output | APEX_QueuingPort_type | queuing port |

  Processing:
  When there is a queuing port identified by *queuingPort_ID*, the presence indicator is `TRUE` and *queuingPort_OUT* is the corresponding queuing port. Otherwise, *present* is `FALSE`.

```
process QUEUINGPORT_CHECKID =
     ( ? Comm_ComponentID_type queuingPort_ID;
       ! boolean present;
         APEX_QueuingPort_type queuingPort_OUT;
     )
     spec (| (| queuingPort_ID -> present
              | { queuingPort_ID -> queuingPort_OUT } when present
              |)
           | (| queuingPort_ID ^= present
              | queuingPort_OUT ^= when present
              |)
           |)
;
```

- process QUEUINGPORT_CHECKNAME
  Interface:

| **Label** | **Nature** | **Type** | **Comments** |
|---|---|---|---|
| *queuingPort_name* | input | Comm_ComponentName_type | queuing port name |
| *present* | output | boolean | presence indicator |
| *queuingPort_ID* | output | Comm_ComponentID_type | queuing port ident. |

  Processing:
  When there is a queuing port named *queuingPort_name*, the presence indicator is `TRUE` and *queuingPort_ID* is the corresponding identifier. Otherwise, *present* is `FALSE`.

```
process QUEUINGPORT_CHECKNAME =
     ( ? Comm_ComponentName_type queuingPort_name;
       ! boolean present;
         Comm_ComponentID_type queuingPort_ID;
     )
     spec (| (| queuingPort_name -> present
              | { queuingPort_name -> queuingPort_ID } when present
              |)
           | (| queuingPort_name ^= present
              | queuingPort_ID ^= when present
              |)
           |)
;
```

- process QUEUINGPORT_RECORD
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *queuingPort_name* | input | Comm_ComponentName_type | queuing port name |
| *queuingPort_Size* | input | QueuingPortSize_type | queuing port size |
| *port_direction* | input | PortDirection_type | source or dest. |
| *message_size* | input | MessageSize_type | message size |
| *queuing_discipline* | input | QueuingDiscipline_type | fifo or priority |
| *queuingPort_ID* | output | Comm_ComponentID_type | queuing port ident. |
| *recorded* | output | boolean | record indicator |

  Processing:
  This service records the queuing port named *queuingPort_name*, and initializes it with the other inputs. The associated identifier is *queuingPort_ID*. The record indicator carries the value `TRUE` when the queuing port is actually recorded; otherwise, it is `FALSE`.

```
process QUEUINGPORT_RECORD =
     ( ? Comm_ComponentName_type queuingPort_name;
         QueuingPortSize_type queuingPort_size;
         PortDirection_type port_direction;
         MessageSize_type message_size;
         QueuingDiscipline_type queuing_discipline;
       ! Comm_ComponentID_type queuingPort_ID;
         boolean recorded;
     )
     spec (| (| { queuingPort_name,queuingPort_size,port_direction,
                  message_size, queuing_discipline } -> recorded
              | { { queuingPort_name,queuingPort_size,port_direction,
                    message_size, queuing_discipline } ->
                    queuingPort_ID } when recorded
              |)
            | (| queuingPort_name ^= queuingPort_size ^= port_direction ^=
                 message_size ^= queuing_discipline ^=
                 recorded
              | queuingPort_ID ^= when recorded
              |)
            |)
;
```

- process QUEUINGPORT_UPDATE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *queuingPort_IN* | input | APEX_QueuingPort_type | queuing port |

  Processing:
  This service updates the queuing port named *queuingPort_IN*.Name in the manager with the status of the queuing port *queuingPort_IN*.

```
process QUEUINGPORT_UPDATE =
     ( ? APEX_QueuingPort_type queuingPort_IN;
     )
```

- process QUEUINGPORT_CHECKCAPACITY
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *full* | output | boolean | manager status |

  Processing:
  This service checks whether or not the queuing port manager is full.

```
process QUEUINGPORT_CHECKCAPACITY =
     ( ! boolean full;
     )
;
```

- process QUEUINGPORT_CHECKCREATED
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *queuingPort_ID* | input | Comm_ComponentID_type | queuing port ident. |
| *created* | output | boolean | answer |

  Processing:
  This service checks whether or not the queuing port identified by *queuingPort_ID* has been already created.

```
process QUEUINGPORT_CHECKCREATED =
     ( ? Comm_ComponentName_type queuingPort_name;
       ! boolean created;
     )
     spec (| (| queuingPort_name -> created |)
           | (| queuingPort_name ^= created |)
           |)
;
```

- process SAMPLINGPORT_CHECKID
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *samplingPort_ID* | input | Comm_ComponentID_type | sampling port ident. |
| *present* | output | boolean | presence indicator |
| *samplingPort_OUT* | output | APEX_SamplingPort_type | sampling port |

  Processing:
  When there is a sampling port identified by *samplingPort_ID*, the presence indicator is

TRUE and *samplingPort_ OUT* is the corresponding sampling port. Otherwise, *present* is FALSE.

```
process SAMPLINGPORT_CHECKID =
     ( ? Comm_ComponentID_type samplingPort_ID;
       ! boolean present;
         APEX_SamplingPort_type samplingPort_OUT;
     )
     spec (| (| samplingPort_ID -> present
              | { samplingPort_ID -> samplingPort_OUT } when present
              |)
           | (| samplingPort_ID ^= present
              | samplingPort_OUT ^= when present
              |)
           |)
;
```

- process SAMPLINGPORT_CHECKNAME
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *samplingPort_ name* | input | Comm_ComponentName_type | sampling port name |
| *present* | output | boolean | presence indicator |
| *samplingPort_ ID* | output | Comm_ComponentID_type | sampling port ident. |

  Processing:
  When there is a sampling port named *samplingPort_ name*, the presence indicator is TRUE and *samplingPort_ ID* is the corresponding identifier. Otherwise, *present* is FALSE.

```
process SAMPLINGPORT_CHECKNAME =
     ( ? Comm_ComponentName_type samplingPort_name;
       ! boolean present;
         Comm_ComponentID_type samplingPort_ID;
     )
     spec (| (| samplingPort_name -> present
              | { samplingPort_name -> samplingPort_ID } when present
              |)
           | (| samplingPort_name ^= present
              | samplingPort_ID ^= when present
              |)
           |)
;
```

- process SAMPLINGPORT_RECORD
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *samplingPort_name* | input | Comm_ComponentName_type | sampling port name |
| *samplingPort_Size* | input | SamplingPortSize_type | sampling port size |
| *port_direction* | input | PortDirection_type | source or dest. |
| *refresh_period* | input | SystemTime_type | mess. refresh period |
| *samplingPort_ID* | output | Comm_ComponentID_type | sampling port ident. |
| *recorded* | output | boolean | record indicator |

  Processing:
  This service records a sampling port named *samplingPort_name*, and initializes with the other inputs. The associated identifier is *samplingPort_ID*. The record indicator carries the value `TRUE` when the sampling port is actually recorded; otherwise, it is `FALSE`.

```
process SAMPLINGPORT_RECORD =
     ( ? Comm_ComponentName_type samplingPort_name;
         SamplingPortSize_type samplingPort_size;
         PortDirection_type port_direction;
         SystemTime_type refresh_period;
       ! Comm_ComponentID_type samplingPort_ID;
         boolean recorded;
     )
     spec (| (| { samplingPort_name,samplingPort_size,port_direction,
                  refresh_period } -> recorded
              | { { samplingPort_name,samplingPort_size,port_direction,
                  refresh_period } -> samplingPort_ID } when recorded
              |)
            | (| samplingPort_name ^= samplingPort_size ^= port_direction ^=
                  refresh_period ^= recorded
              | samplingPort_ID ^= when recorded
              |)
           |)
;
```

- process SAMPLINGPORT_UPDATE
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *samplingPort_IN* | input | APEX_SamplingPort_type | sampling port |

  Processing:
  This service updates the sampling port named *samplingPort_IN*.Name in the manager with the status of the sampling port *samplingPort_IN*.

```
process SAMPLINGPORT_UPDATE =
     ( ? APEX_SamplingPort_type samplingPort_IN;
     )
;
```

- process SAMPLINGPORT_CHECKCAPACITY
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *full* | output | boolean | manager status |

  Processing:
  This service checks whether or not the sampling port manager is full.

```
process SAMPLINGPORT_CHECKCAPACITY =
     ( ! boolean full;
     )
;
```

- process SAMPLINGPORT_CHECKCREATED
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *samplingPort_ID* | input | Comm_ComponentID_type | sampling port ident. |
| *created* | output | boolean | answer |

  Processing:
  This service checks whether or not the sampling port identified by *samplingPort_ID* has been already created.

```
process SAMPLINGPORT_CHECKCREATED =
     ( ? Comm_ComponentName_type samplingPort_name;
       ! boolean created;
     )
     spec (| (| samplingPort_name -> created |)
           | (| samplingPort_name ^= created |)
           |)
;
```

### 8.4.3 Inter-partition communication services

1. APEX Queuing Port

   - process CREATE_QUEUING_PORT*
     Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *queuingPort_name* | input | Comm_ComponentName_type | port name |
| *queuingPort_size* | input | QueuingPortSize_type | port size |
| *port_direction* | input | PortDirection_type | source or dest |
| *queuing_discipline* | input | QueuingDiscipline_type | queuing rule |
| *queuingPort_ID* | output | Comm_ComponentID_type | port ident. |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service is used to create a queuing port (named *queuingPort_ name*). If the request satisfies all the creation conditions, a queuing port identifier (*queuingPort_ ID*) is returned.

(a) The return code value is:
- `INVALID_CONFIG` when there is either not enough space for creating a new queuing port, or no queuing port of the partition is named *queuingPort_ name*, or *queuingPort_ size* is out of range or not compatible with the configuration, or *port_ direction* is invalid or not compatible with the configuration, or *queuing_ discipline* is invalid;
- `NO_ACTION` when a queuing port with the same name has been already created;
- `INVALID_MODE` when the operating mode is `NORMAL`;
- `NO_ERROR` otherwise.

(b) In the case of successful completion, *queuingPort_ ID* denotes an identifier assigned by the OS to the queuing port named *queuingPort_ name*.

```
process CREATE_QUEUING_PORT =
     ( ? Comm_ComponentName_type queuingPort_name;
         QueuingPortSize_type queuingPort_size;
         PortDirection_type port_direction;
         QueuingDiscipline_type queuing_discipline;
       ! Comm_ComponentID_type queuingPort_ID;
         ReturnCode_type return_code;
     )
     spec (| (| { queuingPort_name, queuingPort_size, port_direction,
                      queuing_discipline } -> return_code
              | { { queuingPort_name, queuingPort_size, port_direction,
                      queuing_discipline } -> queuingPort_ID }
                      when (return_code = #NO_ERROR)
              |)
             (| return_code  ^= queuingPort_name
                   ^= queuingPort_size ^= port_direction
                   ^= queuing_discipline
              | queuingPort_ID ^= when (return_code = #NO_ERROR)
              |)
           |)
;
```

- process SEND_QUEUING_PORT*

Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | parameter | ProcessID_type | caller identifier |
| *queuingPort_ID* | input | Comm_ComponentID_type | port identifier |
| *message* | input | MessageArea_type | message address |
| *length* | input | MessageSize_type | message size |
| *timeout* | input | SystemTime_type | time-out value |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service is used to send a message to a queuing port identified by *queuingPort_ID*.

(a) The return code value is:
- `INVALID_PARAM` when the input *queuingPort_ID* does not identify any queuing port, or *timeout* is out of range;
- `INVALID_CONFIG` when *length* is not compatible with the configuration of the specified port;
- `INVALID_MODE` when the specified port is not configured to operate as source port, or preemption is disabled, or the parameter *process_ID* identifies an error handler process;
- `NOT_AVAILABLE` when the input *timeout* value is zero;
- `NO_ERROR` otherwise.

(b) In the case of successful completion,
- when there is enough space in the port's message queue to insert the new message and no other process is waiting to send a message to that port,
  - if no process is waiting on an empty message queue, the message is stored in the message queue.
  - else the first process is removed from the receiving process queue.
    * If the length of the current message does not exceed the maximum length specified by the waiting process, it is retrieved by this process. On the other hand, the current message is not stored in the port's message queue;
    * else only output parameter *length* of the waiting process is initialized with the current message length. The current message is discarded.

    If this process is waiting with a time-out, the affected time counter is stopped. Its state becomes `READY` (except if another process suspended it previously).
- Otherwise, the calling process goes into a waiting state. It is inserted in the sending process queue at the position specified by the queuing rule. A time counter with duration *timeout* is initiated if *timeout* is not infinite.

Finally, process scheduling is performed if preemption is enabled.

```
process SEND_QUEUING_PORT =
```

```
        { ProcessID_type process_ID; }
        ( ? Comm_ComponentID_type queuingPort_ID;
            MessageArea_type message;
            MessageSize_type length;
            SystemTime_type timeout;
          ! ReturnCode_type return_code;
        )
        spec (| (| { { queuingPort_ID, timeout, message, length }
                        -> return_code } when C_return_code
                 |)
              | (| queuingPort_ID ^= timeout ^= message
                   ^= length ^= C_return_code
                 | return_code ^= when C_return_code
                 |)
              |)
        where boolean C_return_code;
    end;
```

- process RECEIVE_QUEUING_PORT*
  Interface:

| **Label** | **Nature** | **Type** | **Comments** |
|---|---|---|---|
| *process_ID* | parameter | ProcessID_type | caller identifier |
| *queuingPort_ID* | input | Comm_ComponentID_type | port identifier |
| *timeout* | input | SystemTime_type | time-out value |
| *maximum_length* | input | MessageSize_type | max mess size |
| *message* | output | MessageArea_type | message address |
| *length* | output | MessageSize_type | message size |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to receive a message from the queuing port identified by *queuingPort_ID*. A return code indicates the issue of the request.

  (a) The return code value is:
  - `INVALID_PARAM` when the input *queuing port_ID* does not identify any queuing port, or *timeout* is out of range;
  - `INVALID_CONFIG` when *maximum_length* is not compatible with the configuration of the specified port;
  - `NOT_AVAILABLE` when the input *timeout* value is zero;
  - `INVALID_MODE` when the specified port is not configured to operate as destination port, or preemption is disabled, or the parameter *process_ID* identifies an error handler process;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion,
  - when the queuing port is not empty,
    – if the length of the first message in the port's message queue does not exceed *maximum_length*, that message is removed from the message queue

and sent ( the output signals *message* and *length* are initialized with the removed message fields);

– else, only the output *length* is initialized with the length of first message. This message is discarded from the message queue.

If there are processes blocked on a full queuing port and if there is sufficient space in the message queue, the first process is removed from the process queue. The message sent by this process is put in the message queue. If the process is waiting with a time-out, the affected time counter is stopped. Its state becomes `READY` (except if another process suspended it).

- Otherwise, the calling process goes into waiting state. It is inserted in the receiving process queue at the position specified by the queuing discipline. A time counter with duration *timeout* is initiated if *timeout* is not infinite.

Finally, process scheduling is performed if preemption is enabled.

```
process RECEIVE_QUEUING_PORT =
     { ProcessID_type process_ID; }
     ( ? Comm_ComponentID_type queuingPort_ID;
         SystemTime_type timeout;
         MessageSize_type maximum_length;
       ! MessageArea_type message;
         integer length;
         ReturnCode_type return_code;
     )
     spec (| (| { { queuingPort_ID, timeout, maximum_length }
                    -> return_code } when C_return_code
              | { { queuingPort_ID, timeout, maximum_length }
                    -> { length, message } }
                    when (return_code = #NO_ERROR)
              |)
           | (| queuingPort_ID ^= timeout ^= maximum_length
                ^= C_return_code
              | return_code ^= when C_return_code
              | length ^= message ^= when (return_code = #NO_ERROR)
              |)
           |)
     where boolean C_return_code;
end;
```

- process GET_QUEUING_PORT_ID*

Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *queuingPort_name* | input | Comm_ComponentName_type | port name |
| *queuingPort_ID* | output | Comm_ComponentID_type | port identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:

This service allows to get the identifier (*queuingPort_ID*) of the queuing port named *queuingPort_name*.

(a) The return code value is:
- `INVALID_CONFIG` when no queuing port is named *queuingPort_name*;
- `NO_ERROR` otherwise.

(b) In the case of successful completion, *queuingPort_ID* denotes the identifier of the queuing port named *queuingPort_name*.

```
process GET_QUEUING_PORT_ID =
     ( ? Comm_ComponentName_type queuingPort_name;
       ! Comm_ComponentID_type queuingPort_ID;
         ReturnCode_type return_code;
     )
     spec (| (| queuingPort_name -> return_code
              | { queuingPort_name -> queuingPort_ID }
                   when (return_code = #NO_ERROR)
              |)
           | (| queuingPort_name ^= return_code
              | queuingPort_ID ^= when (return_code = #NO_ERROR)
              |)
           |)
;
```

- process GET_QUEUING_PORT_STATUS*

Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *queuingPort_ID* | input | Comm_ComponentID_type | port identifier |
| *status* | input | QueuingPortStatus_type | current status |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service allows to get the status of the queuing port identified by *queuingPort_ID*.

(a) The return code value is:
- `INVALID_PARAM` when no queuing port is identified by *queuingPort_ID* in the current partition;
- `NO_ERROR` otherwise.

(b) In the case of successful completion, *status* denotes the current status of the queuing port identified by *queuingPort_ID*.

```
process GET_QUEUING_PORT_STATUS =
     ( ? Comm_ComponentID_type queuingPort_ID;
       ! QueuingPortStatus_type status;
         ReturnCode_type return_code;
     )
     spec (| (| queuingPort_ID -> return_code
              | { queuingPort_ID -> status }
                   when (return_code = #NO_ERROR)
              |)
```

```
                | (| return_code ^= queuingPort_ID
                   | status ^= when (return_code = #NO_ERROR)
                   |)
                |)
    ;
```

2. APEX Sampling Port

- process CREATE_SAMPLING_PORT*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *samplingPort_name* | input | Comm_ComponentName_type | port name |
| *samplingPort_size* | input | SamplingPortSize_type | size of the port |
| *port_direction* | input | PortDirection_type | source or dest. |
| *refresh_period* | input | SystemTime_type | refresh period |
| *samplingPort_ID* | input | Comm_ComponentID_type | port identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to create a sampling port (named *samplingPort_name*). If the request satisfies all the creation conditions, a sampling port identifier (*samplingPort_ID*) is returned.

  (a) The return code value is:
  - `INVALID_CONFIG` when there is not enough space for creating a new sampling port, or no sampling port of the partition is named *samplingPort_name*, or *samplingPort_size* is out of range or not compatible with the configuration, or *port_direction* is invalid or not compatible with the configuration, or *refresh_period* is out of range or not compatible with the configuration;
  - `NO_ACTION` when a sampling port with the same name has been already created;
  - `INVALID_MODE` when the operating mode is `NORMAL`;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion, *samplingPort_ID* denotes an identifier assigned by the OS to the sampling port named *samplingPort_name*.

```
process CREATE_SAMPLING_PORT =
     ( ? Comm_ComponentName_type samplingPort_name;
         SamplingPortSize_type samplingPort_size;
         PortDirection_type port_direction;
         SystemTime_type refresh_period;
       ! Comm_ComponentID_type samplingPort_ID;
         ReturnCode_type return_code;
     )
     spec (| (| { return_code, samplingPort_name, samplingPort_size,
                        port_direction } -> return_code
              | { { return_code, samplingPort_name, samplingPort_size,
                        port_direction } -> samplingPort_ID  }
                        when (return_code = #NO_ERROR)
```

```
            |)
          | (| return_code ^= samplingPort_name
                 ^= samplingPort_size ^= port_direction
                 ^= refresh_period
            | samplingPort_ID ^= when (return_code = #NO_ERROR)
          |)
;
```

- process WRITE_SAMPLING_PORT*
  Interface:

| **Label** | **Nature** | **Type** | **Comments** |
|---|---|---|---|
| *samplingPort_ID* | input | Comm_ComponentID_type | port identifier |
| *message* | input | MessageArea_type | message address |
| *length* | input | MessageSize_type | message size |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service is used to write a message to a sampling port.

  (a) The return code value is:
  - `INVALID_PARAM` when the input *samplingPort_ID* does not identify any sampling port, or the message is too long;
  - `INVALID_CONFIG` when *length* is not compatible with the configuration of the specified port;
  - `INVALID_MODE` when the specified port is not configured to operate as a source;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion, the message is written into the specified port.

```
process WRITE_SAMPLING_PORT =
     ( ? Comm_ComponentID_type samplingPort_ID;
         MessageArea_type message;
         MessageSize_type length;
       ! ReturnCode_type return_code;
     )
     spec (| (| { samplingPort_ID, length, message }
                  -> return_code } |)
           | (| return_code ^= samplingPort_ID ^= length
                ^= message |)
           |)
;
```

- process READ_SAMPLING_PORT*

Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *samplingPort_ID* | input | Comm_ComponentID_type | port identifier |
| *message* | output | MessageArea_type | message address |
| *length* | output | MessageSize_type | message size |
| *validity* | output | boolean | validity indicator |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service is used to read a message from the specified sampling port. The output *validity* indicates whether or not the age of the read message is consistent with the required refresh rate attribute of the port.

(a) The return code value is:
- `INVALID_PARAM` when the input *samplingPort_ID* does not identify any sampling port;
- `INVALID_MODE` when the specified port is not configured to operate as a destination;
- `NO_ERROR` otherwise.

(b) In the case of successful completion,
- if the sampling port is empty, *validity* is set to `FALSE`.
- Otherwise, the last correct message arrived in the port is sent. If its age is consistent with the required refresh period attribute of the port, *validity* is set to `TRUE`, else it is set to `FALSE`.

```
process READ_SAMPLING_PORT =
     ( ? Comm_ComponentID_type samplingPort_ID;
       ! MessageArea_type message;
         MessageSize_type length;
         boolean validity;
         ReturnCode_type return_code;
     )
     spec (| (| samplingPort_ID -> return_code
              | { samplingPort_ID -> validity }
                   when (return_code = #NO_ERROR)
              | { samplingPort_ID -> { message, length } }
                   when C_return_code
              |)
           | (| return_code ^= samplingPort_ID ^= C_return_code
              | validity ^= when (return_code = #NO_ERROR)
              | message ^= length
              | message ^= when C_return_code
              |)
           |)
     where boolean C_return_code;
end;
```

- process GET_SAMPLING_PORT_ID*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *samplingPort_name* | input | Comm_ComponentName_type | port name |
| *samplingPort_ID* | output | Comm_ComponentID_type | its identifier |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service allows to get the identifier (*samplingPort_ID*) of the sampling port named *samplingPort_name*.

  (a) The return code value is:
  - `INVALID_CONFIG` when no sampling port is named *samplingPort_name*;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion, *samplingPort_ID* denotes the identifier of the sampling port named *samplingPort_name*.

```
process GET_SAMPLING_PORT_ID =
     ( ? Comm_ComponentName_type samplingPort_name;
       ! Comm_ComponentID_type samplingPort_ID;
         ReturnCode_type return_code;
     )
     spec (| (| samplingPort_name -> return_code
              | { samplingPort_name -> samplingPort_ID }
                   when (return_code = #NO_ERROR)
              |)
           | (| samplingPort_Name ^= return_code
              | samplingPort_ID ^= when (return_code = #NO_ERROR)
              |)
           |)
;
```

- process GET_SAMPLING_PORT_STATUS*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *samplingPort_ID* | input | Comm_ComponentID_type | sampl port ident |
| *status* | output | SamplingPortStatus_type | its current status |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service allows to get the status of the sampling port identified by *samplingPort_ID*.

  (a) The return code value is:
  - `INVALID_PARAM` when no sampling port is identified by *samplingPort_ID* in the current partition;
  - `NO_ERROR` otherwise.

  (b) In the case of successful completion, *status* denotes the current status of the sampling port identified by *samplingPort_ID*.

```
process GET_SAMPLING_PORT_STATUS =
     ( ? Comm_ComponentID_type samplingPort_ID;
       ! SamplingPortStatus_type status;
         ReturnCode_type return_code;
     )
     spec (| (| samplingPort_ID -> return_code
              | { samplingPort_ID -> status }
                   when (return_code = #NO_ERROR)
              |)
           | (| return_code ^= samplingPort_ID
              | status ^= when (return_code = #NO_ERROR)
              |)
           |)
;
```

## 8.5 Time management

- process TIMED_WAIT*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *delay_time* | input | SystemTime_type | waiting duration |
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service allows the suspension of the current executing process for a minimum amount of elapsed time.

1. The return code value is:
   - `INVALID_MODE` when preemption is disabled or process is error handler;
   - `INVALID_PARAM` when *delay_time* is out of range, or infinite;
   - `NO_ERROR` otherwise.
2. In the case of successful completion,
   - if *delay_time* is zero, the process state is set to `READY`;
   - else the state is set to `WAITING`. A time counter is initiated with duration *delay_time*.

   Process scheduling is performed.

```
process TIMED_WAIT =
     ( ? SystemTime_type delay_time;
       ! ReturnCode_type return_code;
     )
     spec (| (| delay_time -> return_code |)
           | (| return_code ^= delay_time |)
           |)
;
```

- process PERIODIC_WAIT*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service allows the suspension of the current executing process until the next release point in the processor time line that corresponds to the period of the process.

  1. The return code value is:
     - `INVALID_MODE` when preemption is disabled or the process is an error handler, or the requesting process is not periodic;
     - `NO_ERROR` otherwise.
  2. In the case of successful completion, the process state is set to `WAITING`. Its deadline time becomes the sum of the time of the next release point and its time capacity. Process scheduling is performed.

```
process PERIODIC_WAIT =
     ( ! ReturnCode_type return_code;
     )
;
```

- process GET_TIME*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *system_time* | output | SystemTime_type | current system time |
| *return_code* | output | ReturnCode_type | diagnostic |

  Processing:
  This service allows to get the value of the system clock (common to all processors in the module). The return code value is `NO_ERROR`. *system_time* is the current system clock.

```
process GET_TIME =
     ( ! SystemTime_type system_time;
         ReturnCode_type return_code;
     )
     spec (| return_code ^= system_time |)
;
```

- process REPLENISH*
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *return_code* | output | ReturnCode_type | diagnostic |

Processing:
This service allows to update the deadline of the requesting process with its time capacity. The return code value is `NO_ERROR`. A new deadline time value (sum of the current system clock and time capacity value) is set for the current process.

```
process REPLENISH =
     ( ! ReturnCode_type return_code;
     )
;
```

- process START_COUNTER
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |
| *delay* | input | SystemTime_type | initialization value |

Processing:
This service allows to initiate the time counter associated with the process identified by *process_ID*. The initial value of the counter is *delay*.

```
process START_COUNTER =
     ( ? ProcessID_type process_ID;
         SystemTime_type delay;
     )
     spec (| process_ID ^= delay |)
;
```

- process STOP_COUNTER
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |

Processing:
This service allows to stop the time counter associated with the process identified by *process_ID*.

```
process STOP_COUNTER =
     ( ? ProcessID_type process_ID;
     )
;
```

- process RESET_RUNTIME
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_ID* | input | ProcessID_type | process identifier |

Processing:
This service allows to reset the run-time for the process identified by *process_ID*.

```
process RESET_RUNTIME =
     ( ? ProcessID_type process_ID;
     )
;
```

- process STOPALL_COUNTER
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *process_range* | input | array of ProcessID_type | set of process ident |

Processing:
This service allows to stop all the time counters associated with processes specified in *process_range*.

```
process STOPALL_COUNTER =
     ( ? [MAX_NUMBER_OF_PROCESSES]ProcessID_type process_range;
     )
;
```

- process UPDATE_COUNTERS
  Interface:

| Label | Nature | Type | Comments |
|---|---|---|---|
| *timedout* | output | array of boolean | timedout[i] = *true* if the process identified by i should be notified with a "timed out" signal, else timedout[i] = *false* |

Processing:
This service updates the time counters within the partition. So, it decrements all the time counters with positive value. If the value of the counter associated with the process identified by $i$ becomes zero, *timedout[i]* is set to *TRUE*; else *timedout[i]* is set to *FALSE*.

```
process UPDATE_COUNTERS =
     ( ! [MAX_NUMBER_OF_PROCESSES]boolean timedout;
     )
;
```

# 9 Annex C: The implementation architecture of the library

The library is organized as shown in FIG. 25. The files in the elliptic boxes contain original APEX services [Com97b]. The other files in the rectangular boxes, describe additional services that we have defined (e.g. services used for component management). Finally, the file `Types_and_Constants` contains both types and constants defined in [Com97b] in addition to special types required by the added services.

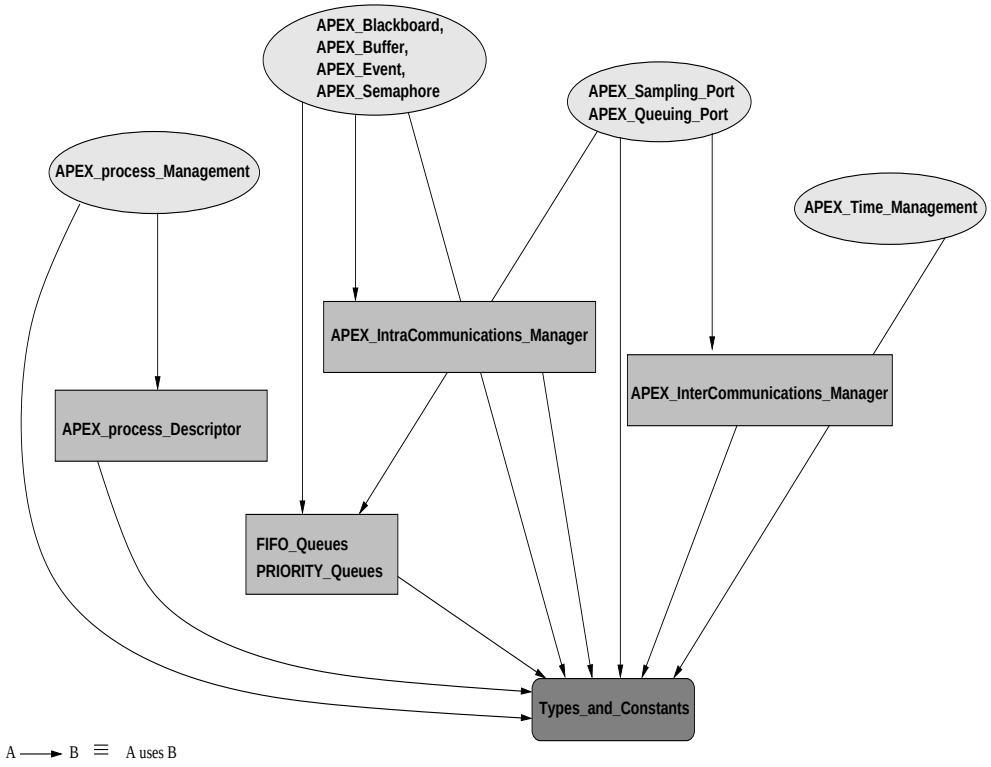



Figure 25: Implementation architecture of the library.

# References


[AD94] R. Alur and D.L. Dill. A theory of timed automata. In *Theoretical Computer Science*, volume 126, pages 183–235, 1994.

[AGS02] K. Altisen, G. Goessler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. In *Journal of Real-Time Systems, special issue on Control Approaches to Real-Time Computing*, volume 23, pages 55–84, 2002.

[BB91] 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.

[BCL00] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation. In *Information and Computation, vol. 163*, pages 125–171, 2000.

[BDL+02] P. Baufreton, F. Dupont, R. Leviathan, M. Segelken, and K. Winkelmann. SAFEAIR: Constructing correct systems in the SAFEAIR project. In *Proceedings of RCS'02 - International workshop on Refinement of Critical Systems: Methods, Tools and Experience, Grenoble, France*, January 2002.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language : design, semantics, implementation. In *Science of Computer Programming, vol. 19(2)*, pages 87–152, November 1992.

[BGG02] L. Besnard, T. Gautier, and P. Le Guernic. SIGNAL v4 - inria version: Reference manual. December 2002.

[BGMR97] P. Baufreton, H. Granier, X. Méhaut, and E. Rutten. The SACRESapproach to embedded systems applied to aircraft engine controllers. In *Proceedings of the 22nd IFAC/IFIP Workshop on RT Programming, Lyon, France*, September 1997.

[Cle96] Paul C. Clements. A survey of architecture description languages. In *8th Int'l Workshop on Software Specifications and Design, Paderborn, Germany*, March, 1996.

[Com97a] Airlines Electronic Engineering Committee. Arinc report 651-1: Design guidance for integrated modular avionics. In *Aeronautical radio, Inc., Annapolis, Maryland*, November 1997.

[Com97b] Airlines Electronic Engineering Committee. Arinc specification 653: Avionics application software standard interface. In *Aeronautical radio, Inc., Annapolis, Maryland*, January 1997.

[Cora] Rational Corporation. See `http://www.rational.com/products/rosert/index.jsp`.

[Corb] Timesys Corporation. See `http://www.timesys.com/prodserv/windows/windows_prod_oview.cfm`.

[CPP+01] 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.

[EAL93] D. Engler, D. Andrews, and D. Lowenthal. Efficient support for fine-grain parallelism. In *Technical report, University of Arizona*, 1993.

[GG99] T. Gautier and P. Le Guernic. Code generation in the sacres project. In *Towards System Safety, Proceedings of the Safety-critical Systems Symposium, SSS'99, Springer, Huntingdon, UK*, February 1999.

[GG02] 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.

[GL99] 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.

[GLM94] T. Gautier, P. Le Guernic, and O. Maffeïs. For a new real-time methodology. In *Technical Report, INRIA (`http://www.irisa.fr/ep-atr/Publis/Annee/1994.html`)*, October 1994.

[GMGW01] D. Goshen-Meskin, V. Gafni, and M. Winokur. SAFEAIR: An integrated development environment and methodology. In *INCOSE 2001, Melbourne*, July 2001.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Publications, 1993.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE, vol.79(9)*, pages 1305–1320, September 1991.

[HHK01] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Embedded control systems development with giotto. In *Proceedings of LCTES. ACM SIGPLAN Notices*, 2001.

[HMP91] T.A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. In *Proceedings of the 18th Annual Symposium on Principles of Programming Languages. ACM Press*, pages 353–366, 1991.

[KL96] A. Kountouris and P. Le Guernic. Profiling of SIGNAL programs and its application in the timing evaluation of design implementations. In *Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems, IEE*, pages 6/1–6/9. HP Labs, Bristol, UK, February 1996.

[KRP+93] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. A practitioner's handbook for real-time analysis: Guide to rate monotonic analysis for real-time systems. In *Kluwer Academic Publishers*, 1993.

[La01] Edward A. Lee and al. Overview of the ptolemy project. In *Technical Report UBC/ERL M01/11, University of California at Berkeley*, March 2001.

[Lee00] Edward A. Lee. What's ahead for embedded software? In *IEEE Computer*, pages 18–26, September 2000.

[LGLL91] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. In *Proceedings of the IEEE, 79(9)*, pages 1321–1336, September 1991.

[MBLL00] 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.

[Sif77] J. Sifakis. Use of petri nets for performance evaluation. In *Proceedings of the 3rd International Symposium on Modeling and Evaluation. IFIP, North Holland*, pages 75–93, 1977.

[Sif01] J. Sifakis. Modeling real-time systems - challenges and work directions. In *EMSOFT'01, Tahoe City. Lecture Notes in Computer Science 2211*, October 2001.

[SR98] Bran Selic and James Rumbaugh. Using UML for modeling complex real-time systems. In *ObjecTime Limited (available at* `http://www.objectime.com`*)*, March 1998.

[Ves97] S. Vestal. Metah support for real-time multi-processor avionics. In *IEEE Workshop on Parallel and Distributed Real-Time Systems*, April 1997.

[WBC+00] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of ESTEREL for real-time embedded systems. In *Proceedings of CASES'2000, San Jose*, pages 2–8, November 2000.

[Yov97] S. Yovine. KRONOS: A verification tool for real-time systems. In *Software Tools for Technology Transfer, 1(1+2)*, pages 123–133, December 1997.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399