A Generic Formal Model for RTOS [2]

Synchronous Approach for Rapid Virtual Prototyping

Abdoulaye Gamatié

CNRS / LIFL (UMR 8022)
INRIA Lille - Nord Europe
Villeneuve d'Ascq, France
abdoulaye.gamatie@lifl.fr

ABSTRACT. Real-time operating systems (RTOS) play a central role in the correct and efficient management of computing resources for applications with stringent real-time constraints. As a consequence, addressing them in an adequate way during the development of real-time systems is highly important. This paper proposes a high-level approach to the design of such systems. The rationale of this approach consists in using the synchronous approach to define a general modeling framework making possible RTOS-based design according to three major standards: APEX, Posix and OSEK, respectively dedicated to avionic, general-purpose and automotive applications. The suggested solution specifies generic models of RTOS services and executive entities, covering the standards. The main advantage of these models is that they favor a rapid virtual prototyping of real-time systems, with access to formal validation techniques.

RÉSUMÉ. Les systèmes d'exploitation temps réel (acronyme anglais, RTOS) jouent un rôle central dans la gestion correcte et efficace des ressources d'exécution pour les applications avec des contraintes temps réel. Par conséquent, il est très important de savoir les aborder de façon adéquate lors du développement des systèmes temps réel. Ce papier propose une approche à haut niveau pour la conception de tels systèmes. L'idée de cette approche consiste à utiliser l'approche synchrone pour définir un cadre général de modélisation rendant possible la conception orientée RTOS, selon trois standards : APEX, Posix et OSEK, respectivement dédiés aux applications avioniques, générales et automobiles. La solution suggérée spécifie des modèles génériques de services d'un RTOS et aussi d'entités d'exécution couvrant les standards. Le principal avantage de ces modèles est qu'ils favorisent un prototypage virtuel rapide de systèmes temps réel, avec un accès à des techniques de validation formelle.

KEYWORDS : Real-time systems, RTOS, modeling, synchronous approach, APEX, Posix, OSEK, formal validation, Signal language

MOTS-CLÉS : Systèmes temps réel, RTOS, modélisation, approche synchrone, APEX, Posix, OSEK, validation formelle, langage Signal

2. The current paper partly relies on earlier studies funded by the European project IST SafeAir (Advanced Design Tools for Aircraft Systems and Airborne Software) - Convention n^o 1 00 C 0149 00 31307 00 5.

1. Introduction

A real-time system is a computer system expected to execute as fast as possible to produce a response upon its environment's requests within a bounded delay. The environment often includes a physical process, e.g., a nuclear power plant, sending requests to its associated controlling computer system via sensors. In such a situation, the system is particularly required to satisfy both response-time and functional correctness constraints. Real-time operating systems (RTOS) are key components in the design of real-time systems since they allow applications to get access to computer resources. For modern applications, which are increasingly complex and performance-demanding, it is important to adequately deal with issues such as concurrency, memory or time management. RTOS offer concurrent programming paradigms including executable entities running in common or separate address spaces. They also propose services for scheduling, synchronization, communication, time and memory management. The reliable and efficient design of real-time systems taking into account all these aspects has become more challenging than ever.

Among promising solutions to the design of real-time systems, we mention *model-based* approaches, which make it possible to address design issues at different abstraction levels, thus enabling earlier and flexible design decisions. These approaches also facilitate the design by favoring genericity, modularity and reusability. They are often associated with formal techniques for correctness analysis and predictability. They are now widely adopted in industry for the development of safety-critical real-time systems.

Our proposition: Synchronous models of RTOS components. This paper proposes an integration of different RTOS features in the formal modeling framework of the synchronous language Signal [7, 2], according to three standards: APEX [1], Posix [10] and OSEK [9], respectively dedicated to avionic, general-purpose and automotive applications. This proposition is built upon a previous work on APEX modeling [3]. In Section 2, we discuss the three standards and compare them. In Section 3, the modeling of our generic RTOS components in Signal is shown. Such components are usable in the top two layers in Figure 1: *application level* and *API*. They consist of executive entities and services for scheduling, synchronization, communication, time and memory management. Section 4 exposes all formal analysis facilities applicable to the resulting models. The goal is to enable a rapid virtual prototyping permitting earlier design space exploration. Finally, Section 5 gives concluding remarks.

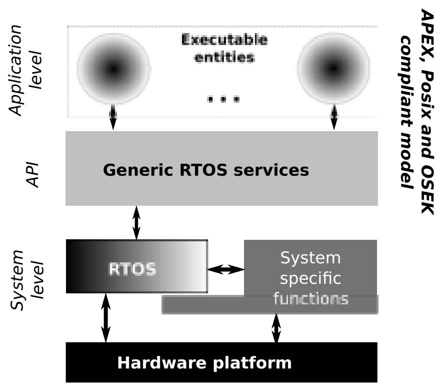

Figure 1. *Considered system architecture.*

2. APEX, Posix and OSEK standards

2.1. Overview

APEX. The APEX (*APplication EXecutive*) standard specified in the ARINC 653 series [1] defines an RTOS interface for applications to execute on integrated modular avionics (IMA) architectures. IMA is an alternative solution to the high cost of federated architectures in which system functions are separately executed on their own computer system. IMA allows several functions with different criticality to execute on the same computer. It ensures a safe allocation of shared resources without fault propagation. This is achieved via a *partitioning* of resources *w.r.t.* available time and memory budget. A *partition* is a logical allocation unit resulting from a functional decomposition of the system. It is allocated a processor for a fixed time window within a global time frame maintained by an operating system (OS). Partitions communicate asynchronously via *ports* and *channels*. They are composed of *processes* running concurrently to fulfill the functions associated with their partition. Each process is uniquely characterized by information, e.g., period, priority, or deadline time, useful to the partition-level OS, which is responsible for the correct execution of processes within a partition. The scheduling policy for processes is priority preemptive. Communications between processes are achieved by three basic mechanisms: *buffers*, *events* and *blackboards*, while *semaphores* enable synchronizations. APEX defines services for communication between partitions and processes, synchronization of processes, and management of partitions, process, time and memory.

Posix. The real-time extension of Posix (*Portable operating system interface*), referred to as RT-Posix [10] defines a number of profiles that enhance the basic standard with capabilities for the development of real-time and embedded applications. The profiles propose several RTOS services among which are scheduling services. The executive entities are *processes* and *threads*. A process is composed of threads and plays a similar role as an APEX partition *w.r.t.* APEX processes. Different scheduling strategies can be envisaged: *sched_fifo* assumes a *First In First Out* policy, *sched_rr* considers a *Round Robin* policy and *sched_other* assumes other implementation-specific policies. Threads are schedulable from different perspectives. At the *global* level, they are managed irrespective of their membership to processes whose scheduling parameters are ignored. At the *local* level, the scheduling of threads is restricted to the process they belong to. The last perspective combines the previous two. RT-Posix defines services for synchronization between executive entities, e.g. by using semaphores; resource access via mutual exclusion; communications through message queues; and time management.

OSEK. The OSEK (*Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug*) standard [9] is dedicated to embedded electronics in the automotive domain. Similarly to APEX and Posix, it specifies a set of RTOS services. Two kinds of executive entities are distinguished: *interrupts* and *tasks*. Their scheduling is based on static priorities. Interrupts have higher priorities than tasks. They are managed by the hardware. Thus, their priorities strongly depend on the considered implementation platform. Tasks are handled by considering a scheduler as for APEX and Posix executive entities. A very popular OSEK-compliant RTOS is VDX (*Vehicle Distributed eXecutive*), which implements the services specified by the standard for multi-task applications. During the development of an application, all the objects must be statically allocated. OSEKtime is another variant of the OSEK standard dedicated to time-triggered execution.

2.2. Discussion and comparison

The services specified in the different standards aim at the same RTOS functionalities: entity scheduling, synchronization, communication, time and memory management. In essence, they are actually very similar. Of course, there could be some minor differences, going from interface specifications to implementation choices. Our solution proposes generic service models that can be easily instantiated according to implementations of mentioned standards (their minor differences are considered via a black-box vision).

On the other hand, we observe that each standard defines its proper notion of executive entity. However, a strong analogy exists between the visions adopted by APEX and Posix. OSEK considers two entity models without any containment hierarchy contrarily to APEX and Posix. In order to cover all the three standards, we have to propose entity models that can be either hierarchical or not, and can execute concurrently. The way these entity models have to be managed will depend on the scheduling paradigms supported by each standard. In Posix the scheduling of processes relies on priorities while in APEX the corresponding entities, i.e. partitions, are scheduled based on a time sharing. In addition, the scheduling of Posix threads is possible from different perspectives (local, global and mixed) while in APEX, the scheduling of processes is only local. In order to support the scheduling of Posix threads, APEX processes and OSEK tasks and interrupts, fixed and dynamic priorities must be supported. Finally, all other scheduling paradigms should be made possible: time-triggered, round-robin and FIFO.

3. Generic RTOS component models

The generic RTOS components proposed in this section are illustrated in the synchronous language Signal [7, 2]. Signal basically assumes a *logical* notion of time, defined according to partial order and simultaneity of events observed during system executions. It handles unbounded series of typed values (e.g., **integer, boolean**), called *signals* and denoted as **x** in the language. At a given logical instant, a signal may be either present, at which point it holds a relevant value; or *absent* and represented by $\perp$ at this point. There is a particular type of signals called **event**. A signal of this type is *true* whenever it is present. The set of instants at which a signal **x** is present is termed its *clock* and written $\hat{\mathbf{x}}$, of type **event**. Two different signals **x1** and **x2** with the same clock are said to be *synchronous*, noted as $\mathbf{x1} \hat{=} \mathbf{x2}$. A *process* is a system of equations defining constraints over the clocks and values of signals. A *program* is a process. Signal relies on six primitive constructs used to define processes modularly. Among these constructs are the *parallel composition* **P | Q** denoting the union of equations in processes **P** and **Q**; and the *local declaration* **P where x**, restricting the scope of a signal **x** to a process **P**.

3.1. RTOS services

Our generic service model illustrated in Figure 2, distinguishes two levels. Even though it is represented by a unique hierarchical box, it should be understood from different perspectives: *abstract* and *detailed views*. The upper level (gray background) captures an abstract view of a service by specifying its behavioral properties in a very general way. This is done by describing the interface properties of the service: identifiers and types of parameters, data-dependencies and clock constraints between inputs and outputs. This level of detail is rich enough to enable relevant formal analysis in Signal. The inner level (dark background) of the model in Figure 2 represents a detailed view of the service that specializes the generic model as an implementation according to standards by taking into account their minor differences. So, from a single abstract service model, different specific models are obtained by detailing the internal part of the model in addition to interface properties.

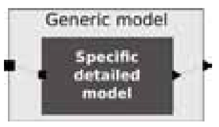

Figure 2. *Generic service.*

In order to illustrate the above principle, let us consider a service associated with semaphores to perform a "wait" request. In Posix, such a service is known as *sem_wait()* while in APEX it is denoted by the service *wait_semaphore*. The following Signal process named **Generic_Wait_Semaphore** is a generic model for the wait service:

```
01: process Generic_Wait_Semaphore =
02:   { EntityID_type entity_ID; }
03:   ( ? ResourceID_type semaphore_ID; SystemTime_type timeout;
04:     ! ReturnCode_type return_code; )
05: spec
06:     (| (| return_code ^= when C_return_code
07:          | semaphore_ID ^= timeout ^= C_return_code  |)
08:       | semaphore_ID, timeout --> return_code when C_return_code  |)
09:     where boolean C_return_code;
10: end;
11: pragmas C_CODE "&o1 = wait_semaphore(&i1, &i2)" end pragmas;
```

Generic_Wait_Semaphore is parametrized with the identifier of the requesting entity (line **(02)**), used as information when the entity is suspended upon the request. The inputs and output parameters of the service are declared at lines **(03)** and **(04)** respectively. This process mainly expresses interface properties, introduced by the keyword **spec** in Signal. For instance, line **(06)** specifies the logical instants at which a **return_code** signal is produced on an invocation of the service. **C_return_code** is a local Boolean signal that must be *true* whenever a return code is produced: it is expressed via the "**when**" operator of Signal. Line **(07)** states that it occurs (with either *true* or *false* values) whenever there is a wait request, denoted by the simultaneous occurrence of input signals. Line **(08)** specifies a data dependency between inputs and output when a **return_code** occurs. These interface properties, characterizing the upper level of the service model, are expressive enough to enable relevant dependency and clock analysis.

Now, in order to make a link with the inner level of the model, representing some specific implementations of the service, the **pragma** notion of Signal is used. Here, a **pragma** is a statement to be taken into account during automatic code generation from a component model. Line **(11)** states that the C code implementing the detailed behavior of **Generic_Wait_Semaphore** is given by a **wait_semaphore** function defined elsewhere. This function refers to the inputs and output of the service model respectively via the **&i1**, **&i2** and **&o1** encodings. It is either user-defined or an *intellectual property* provided in a library of C code for RTOS services. The same holds for C++ or Java code from Signal.

3.2. Executive entities

The generic model of executive entities is depicted by Figure 3. It comprises three basic building blocks: a *local controller*, *local resources* and a set of *actions* that achieve together and in a modular way the functionality associated with the entity. According to the considered perspectives for scheduling, the controllers and actions play various roles as explained below. The role of the controller is to manage the actions to be performed by an entity. It may be a simple finite state machine describing the execution flow in an elementary executive entity such as a Posix thread or an APEX process. It may also be a complex scheduler in a composite entity where the actions are themselves executive entities. In such a case, it plays the role of Posix process or APEX partition level operating system. Finally, at another hierarchical level in a composite entity, where actions can be either Posix processes or

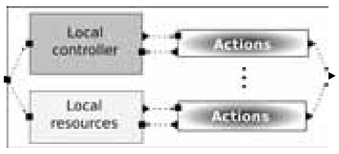

Figure 3. *Generic entity.*

ARINC partitions, the controller will describe a suitable scheduling paradigm. The local resources consist of all data required within an entity to perform the actions. They include inputs and local data in an elementary entity, local communication and synchronization resources in a composite entity, e.g., events and semaphores. In elementary entities, an action denotes the minimum subset of statements to execute atomically, e.g., a service invocation. So, it defines the preemption level. In composite executive entities, actions become potential hierarchical entities.

As an illustration, let us consider a 2-level composite entity, e.g., a Posix process or an APEX partition. The following Signal processes are component templates combined with the parallel composition operator to obtain a composite or an elementary executive entity (for the sake of simplicity, interface properties are omitted):

```
process Main_Controller =
  { EntityID_type entity_ID; ... }
  ( ? EntityID_type Active_entity_ID; real dt;
    ! EntityID_type Active_SubEntity_ID; [NB_Entity_ID] boolean timedout;) ...
process Sub_Controller =
  { EntityID_type entity_ID; ... }
  ( ? EntityID_type Active_entity_ID; [NB_Actions]boolean timedout;
      [NB_Actions] Control_type info;
    ! BlockID_type Active_action_ID;) ...
process Action =
  { ActionID_type action_ID; ... }
  ( ? ActionID_type Active_action_ID; InputData_type r1,...,rk;
    ! Control_type info; SystemTime_type dt; OutputData_type o1,...,oj;) ...
```

In the **Main_Controller** process, the input **Active_entity_ID** denotes the identifier of the selected composite entity to run (when it is equal to **entity_ID**). Whenever this entity executes, **Main_Controller** selects an active sub-entity denoted by the output **Active_SubEntity_ID**. The input signal **dt** denotes a duration information corresponding to sub-entity execution. It is used to update time counters. The output signal **timedout** reflects the current status of the time counters within the composite entity.

In **Sub_Controller**, whenever the input **Active_entity_ID** identifies a sub-entity, the corresponding actions are executed. These actions are identified by **Active_action_ID**. Since **Sub_Controller** expresses the control flow of a sub-entity, it may require some feedback information, denoted by the **info** input signal, from the performed actions. It also requires counter status to decide what to do when the sub-entity has been already suspended on a blocking service request such as **Generic_Wait_Semaphore**.

In a similar way, atomic actions (modeled by the **Action** process) are executed whenever they are selected by a **Sub_Controller** process. In that case, they take input data and produce output data as well as the other information required by the different controllers: **dt** for counter management and **info** for sub-entity control flow management.

At the **Main_Controller** level, local resources are represented by declarations of communication and synchronization mechanisms, and data required by functions. At the **Sub_Controller** level, they mainly consist of input data for elementary entities.

From a practical point of view, the generic models of RTOS services and executive entities presented above could be obtained from a straightforward extension of the component library described in [3]. This library has been fully implemented in the Signal environment, Polychrony, for the APEX standard.

4. Formal analysis and simulation for earlier validation

Polychrony offers a set of facilities usable on designed models for formal verification and analysis, as well as automatic code generation in C, C++ or Java.

Reactivity, determinism, mutual exclusion, deadlock and safety. The Signal compiler implements a powerful formal calculus that allows the designer to address the functional properties of an application model. Beyond the syntactic and type checking of general-purpose languages compilers, it suitably deals with key properties such as reactivity and determinism. For this purpose, it focuses on the clock constraints inherent to a given program. Typically, reactivity is ensured by checking the absence of empty signal clocks in the program, which means that there is no signal that never occurs. Reactivity is a major characteristic of real-time systems since they are expected to react whenever their connected environment solicits them. Functional determinism is another crucial characteristic of real-time systems, which can be guaranteed by checking that a Signal program satisfies the so-called endochrony property. Such a property relies on a specific clock hierarchy resulting from the clock analysis performed by the compiler.

Further important design aspects such as mutual exclusion can be also addressed by analyzing clocks. This is possible by asking the compiler to partition the signals of a program according to the fact that they never occur at the same instants. In other words, they have mutually exclusive clocks. Such an information is useful to ensure absence of resource access conflicts among different executive entities. For instance, on concurrent write in a buffer, if the written data are represented by signals with mutually exclusive clocks, then no additional synchronization will be required for the buffer access.

In addition to clock-based analysis, the compiler also focuses on data-dependencies to verify that no instantaneous cyclic definitions exist in the program in order to avoid execution deadlocks. Finally, more advanced functional properties are checked by model-checking with the Sigali tool [8], connected to Polychrony in a seamless way. Usual techniques like reachability and liveness analyses are made possible to prove the safety of Signal models of real-time application *w.r.t.* given user-defined properties.

Performance analysis. Temporal properties, which are of high importance in real-time systems, are also analyzable by using a performance analysis technique [6]. Basically, it consists of formal transformations of a Signal program **P** into another program $\mathcal{T}(\mathbf{P})$ corresponding to a quantitative temporal interpretation of the initial one. $\mathcal{T}(\mathbf{P})$ serves as an observer of **P** in which the temporal properties of interest are specified. In [4], this technique is applied to a realistic avionics application modeled in Signal with APEX to illustrate a performance analysis.

Functional simulation. The mathematical foundations of the Signal language confers to programs a non ambiguous semantics that is trust-worthily manipulated to define correct-by-construction program transformations. The automatic code generation provided by Polychrony benefits from this principle. It enables to produce simulation code in general-purpose languages such as C or Java. The code can be either monolithic or distributed. It therefore offers the opportunity for intensive functional simulations that exhibit system behaviors according to considered design choices. This serves as a basis for the designer to assess and improve the designed application models.

5. Concluding remarks

In this paper, we have shown how features of different RTOS standard are integrated in a unique formal modeling framework for the rapid virtual prototyping of real-time applications. The considered standards are APEX, Posix and OSEK. Our proposition is built upon a previous study about the synchronous modeling of avionics applications in Signal. Here, the resulting generic models serve together with the synchronous technology to address the flexible and reliable design of real-time applications. They offer an abstraction level allowing one to cope with system design independently from specific RTOS features. Different RTOS APIs are easily reusable and interchangeable from a unique generic model. To the best of our knowledge, only very few design frameworks for real-time systems provide such capabilities. As an example, we can only mention the SynDEx approach that considers a form of generic RTOS services adapted to its associated environment [5]. On the other hand, the description of our models in Signal gives access to a rich formal tool-set suitable for the trustworthy analysis and simulation of designs. Integrating all these aspects in a design process became necessary with the growing complexity of safety-critical applications. The main perspective of this work concerns the full implementation of the whole service models in Polychrony, the Signal design environment (note that we already have a large part of these models via [3]).

6. References

[1] Airlines Electronic Engineering Committee. ARINC specification 653: Avionics application software standard interface., Janvier 1997. Aeronautical radio, Inc., Annapolis, Maryland.

[2] A. Gamatié. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*. Springer, New York, 2009.

[3] A. Gamatié and T. Gautier. Synchronous modeling of modular avionics architectures using the signal language. Research Report RR-4678, INRIA, December 2002.

[4] A. Gamatié, T. Gautier, P. Le Guernic, and J.-P. Talpin. Polychronous design of embedded real-time applications. *ACM Trans. on Software Engineering and Methodologies*, 16(2):9, 2007.

[5] T. Grandpierre, C. Lavarenne, and Y. Sorel. Modèle d'exécutif distribué temps réel pour SynDEx. Research Report RR-3476, INRIA, August 1998. (in french).

[6] 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*, pages 6/1–6/9. HP Labs, Bristol, February 1996.

[7] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design, World Scientific*, 12(3):261–303, June 2003.

[8] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of Discrete-Event Controllers based on the Signal Environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4):325–346, October 2000.

[9] OSEK Group. OSEK/VDX-Operating System Specification 2.2.2, July 2004. *http://www.osek-vdx.org*.

[10] The Institute of Electrical and Electronics Engineers. IEEE Standard for Information Technology- Standardized Application Environment Profile (AEP)-POSIX Realtime and Embedded Application Support, 2004. *http://ieeexplore.ieee.org/xpl/standardstoc.jsp?isnumber=29578*.