A Method and a Technique to Model and Ensure Timeliness in Safety Critical Real-Time Systems

Christophe Aussaguès*+
Vincent David*

*LETI (CEA - Advanced Technologies)
DEIN - CEA/Saclay
F-91191 Gif sur Yvette Cedex - France
e-mail: {chris, david}@albatros.saclay.cea.fr

+University of Marseilles - LIM Laboratory
F-13288 Marseille - France

Abstract

The main focus of this paper is the problem of ensuring timeliness in safety critical systems. First, we introduce a method and its associated technique to model both real-time tasks and the timeliness ensuring concern when tasks are executed in parallel. This approach is based on formal aspects of our real-time tasks model and on the definition of the synchronized product operator on the tasks. Real-time tasks are equivalent to their state-transition diagrams and the operator allows us to compose the diagrams of a set of tasks to represent their interactions. The operator is then used to map the tasks to a system of linear constraints to determine the schedulability of the tasks and deduce a system load upper bound. An illustration of our technique on a safety critical study case is presented in which the timeliness property can be achieved for the real-time set of tasks executed in parallel on the same processor. We also introduce how this work can be applied to the multiprocessor case.

Keywords: safety critical systems, real-time, timeliness, modeling, analysis.

1. Introduction

In the real-time domain, the design and the implementation of safety critical systems are still challenging concerns. Real-time constraints induce not only the presence of timing constraints, such as deadlines dictated by the environment, but also an increasing application complexity, such as the management of different time-scale entities [1]. These constraints complicate the solution of many problems that are well understood if no timing constraint has to be considered.

The real-time domain where raised problems are in full extent, is the one of controlling time critical complex systems. They can generally be found in embedded systems and more precisely in nuclear plants (control, protection and alarm subsystems), spatial and aeronautic systems or robotics. In these different fields, safety standards have been dictated, such as the ISO/IEC-880 standard for safety of nuclear power plants, the DO-178 for airborne systems. An overview of these standards can be found in [2].

Safety critical systems have hard timing requirements. Logical and temporal correctness must be taken into account as soon as possible [3]: missing a deadline of a task will have catastrophic consequences and must be considered as a major fault. The concern of dependability also increases the efforts that have to be done in the design and validation steps of a safety critical system. Adequacy between the dependability concerns and the designing, the implementing or the executing methods has to be proved, thanks to determinism and predictability properties.

Two common approaches to design and implement safety critical applications can be identified. Two main implementing models are possible, a « sequential » and a « parallel » approach, which have different properties concerning the performance and safety aspects. The former approach corresponds to the loop programming techniques. The existing safety critical applications are still mainly realized with these ad-hoc techniques. They can ensure a high level of dependability, but if they are used to design complex applications, where there are numerous different time-scales, the system will be considerably oversized. The « synchronous approach » (see [4]) is a well-defined implementation of these techniques. On the other hand, the parallel implementation of multitasking matches the historical Dijkstra's works and more recently the works on CSP (see [5]) from which derive the Occam and Ada languages. These approaches allow to design and use real-time monitors to execute applications. The more significant benefit of these approaches is the high level of application complexity they afford. But it is connected with an important drawback that is the low level of dependability they ensure. Indeed, determinism and predictability properties are no more guaranteed and analytic verifications are more difficult: an exponential number of tests are necessary to do analyses of such complex systems.

Recent works in the real-time domain are numerous. They are mainly based on parallel implementing models and are dealing with problems described above. Works are also dealing with ensuring the timeliness property thanks to the definition of an implementation of real-time task and communication scheduling algorithms [6] [7] [8] [9]. Kopetz's works (see [10]) deal with the time-triggered approach to design fault tolerant real-time systems based on a static pre-run-time scheduling. Its implementation corresponds to a « sequential » execution model (in our preceding classification). Fault-tolerance mechanisms and parallelism model, developed inside one processor, do not match the focused problem for designing a safety critical real-time system that will be oversized.

Insertion of parallelism techniques (e.g. multitasking) in design and implementation steps of safety critical systems, while still ensuring some dependability properties, such as timeliness, is the main focus of our approach. The OASIS approach (see [11] and [12]) proposes rules and methods for safety-critical application engineering. It allows to design and to implement parallel programs with fully deterministic and predictable behaviors and thus to guarantee some specified properties of dependability. In this paper, we will focus on describing a modeling method and its associated technique (the synchronized product operator) to ensure timeliness.

The paper is organized as follows. In section 2, we introduce the OASIS approach and its real-time task model. We then define and analyze in section 3 the dependability properties that we consider in this paper. Finally, the way to ensure the timeliness property is the focus of the section 4. In this section, we introduce the definition and the construction of the task synchronized product. To illustrate our technique, a safety critical case study is finally introduced in section 5.

2. Description of the OASIS model

2.1. Main principles

In the OASIS approach, an application is viewed as a set of task, named isochronous synchronized active objects, that interact to achieve their goals and real time is managed by a time-triggered approach.

A Time-Triggered system. As Kopetz [13] does, we split system architectures into two distinct categories, depending on their way of handling time during execution. Real-time systems have commonly an event-triggered architecture: occurrences of significant events (such as alarms) initiate system activities and lead the execution of the tasks. On the contrary, in a time-triggered (or TT) architecture, the system will observe the environment and initiate its processings at recurring predetermined points of the globally synchronized time. Then, we can better control the time-scale of each task in the system, and their observations. Because this approach allows us to ensure system determinism, we think that it is best suited for safety critical systems and we will not discuss it further in this paper.

With each task ω in the system, we associate a real-time clock H_ω that represents the set of physical instants where input and output data can be observed. The joining of all clocks H_ω for each ω in Ω, the entire set of tasks of the application, is the definition of the system real-time clock H_Ω. The H_Ω clock includes all the observable instants of the system and is global, fundamental to the whole system. In our approach, we only manipulate regular real-time clocks, i.e. clocks where there is always the same amount of time between two consecutive instants. We then construct the smallest regular clock that includes all H_Ω instants. From now and forward, we consider that these two clocks are equivalent and we note both H_Ω. We deduce the regular clock H_ω from the global clock, i.e. H_Ω, by exhibiting a factor K_ω and a shifting Δ_ω such as: $H_\omega = K_\omega * H_\Omega + \Delta_\omega$.

Each H_ω can be manipulated thanks to the specific advance instruction (*ADV*) we define. The instruction « *ADV*(*k*) » in the task code means that the next activation instant will be its current instant increased by *k* H_ω periods. Thus, we can declare the future instants of the tasks (i.e. activation instants) and both earliest start date and latest end date (i.e. deadline) of each processing. For example, in the following code series { *processing_1* , *ADV(sta)* , *processing_2* , *ADV(end)* }, *sta* in fact represents the earliest start date and *end* the latest end date of *processing_2*. So, we introduce the formal duration of the processing '*processing_2*' as being the value of *end-sta* periods in H_ω units. Hence, however tasks are implemented and executed, their formal duration is unvarying (isochronous property). It contributes to ensure the independence of the target computer system and the absolute predictability of the real-time system.

Isochronous synchronized active objects. The OASIS approach is an object-based approach. Basic system entities or tasks are isochronous synchronized active objects: processing and data are encapsulated in the same structure, and data have only one writer (the owner task). As mentioned above, each task has its own real-time clock deduced from H_Ω, so that different time-scale tasks can coexist. No assumption about task nature is made. We are indeed able to describe every algorithmic complexity of the task code. For example, sporadic and periodic tasks are particular cases and can be easily described in our real-time task model. Moreover, for the observers, formal execution times of all task processings are independent of the target computer. They are equal for the observers to the real time interval between their earliest start dates and their latest end dates. To ensure determinism and predictability, we suppose first that the number of tasks is off-line fixed so that there is no dynamic creation of tasks. Secondly, we suppose that duration of all processings is bounded by known limits. A way to achieve it, is to suppose that all iteration instructions are bounded by known limits and that there doesn't exist any recurrent or divergent algorithmic structure. These assumptions are verified in our application domain (protection automata for alarms or emergency stop procedure of nuclear power plants). From now, we then suppose we are able to exhibit processing upper bounds for each real-time task.

Communication mechanisms. There are two kinds of communication in the OASIS approach [14]. On the one hand, we define an implicit communication mechanism, temporal variables. Each temporal variable is a real-time data flow: their values are stored and updated by the unique writer, the owner task ω, at each instant of H_ω. The system will update (i.e. copy or change if it is a processing ending date) the variable values at each instant of H_ω. Even if a fail occurs, the system will copy the latest correct value of the variable (from the preceding period). This definition of the temporal variables seems similar to the works of H. R. Callison on the Time-Sensitive Object model [15].

On the other hand, we have an asynchronous message passing mechanism, i.e. an explicit communication mechanism. Beyond common message attributes (content, recipient, etc.), the sender indicates the message type and its visibility instant. This instant is the date beyond which the message is visible by the recipient task. Each task has message storage queues. A message queue can heap only messages with the same type and has a lapsing delay (beyond which a specific procedure, such as deletion, can be executed). To achieve determinism, messages are totally ordered in the message queues.

2.2. Formal aspects of the real-time task model

We introduce a formal notation to describe real-time tasks. The notation is based on the OASIS task model. Whatever the task model, real-time tasks generally have some specific features. Their activation laws are commonly sporadic. Periodic activation laws are then a subcase of the sporadic laws. So, real-time tasks have no termination and can be represented as a main « for ever » loop that encapsulates all the task processings. They have always timing constraints on their code execution. In the OASIS approach, tasks have at least one *ADV* instruction to describe their timing constraints (see 2.1. section). Aperiodic tasks are not considered in our model, because we do not consider them as critical in our safety critical domain.

We can define, through the *ADV* instruction, formal activation instants of a task. If we consider the example in 2.1, between the instant *sta* and the instant *end*, the task is formally at the instant *sta*. The formal duration of p_2 is 2 H_Ω-periods. We illustrate it in the following time diagram (see Figure 1).

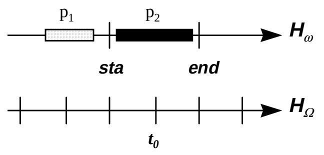


Figure 1. A time diagram

We suppose that the task ω has a temporal variable X. Whatever its current value is between the *sta* and *end* instants, i.e. whatever the p_2 processing computes on X, if another task observes the value of X at the t_0 instant, it will see $X(t_0) = X(sta)$. For the same ground, if ω receives a message with its visibility instant equal to t_0, the task ω will not consume it before the *end* instant.

Code description of real-time tasks. For coding and programming applications with the OASIS approach, we need to write the specific primitives of the real-time task model (the *ADV* and *SEND* instructions, etc.). Languages, like C and Ada, have qualified compilers and are used for programming real-time applications. We have defined a ΨC (resp. ΨAda) dialect above the C language (resp. the Ada language). This dialect is a semi-formal language that contains all the formal aspects of the

OASIS approach. Since it is not the subject of the paper we will not describe it further. Nevertheless, we use in this paper a pseudo-code notation similar to a reduced Ψ-dialect for describing real-time task code.

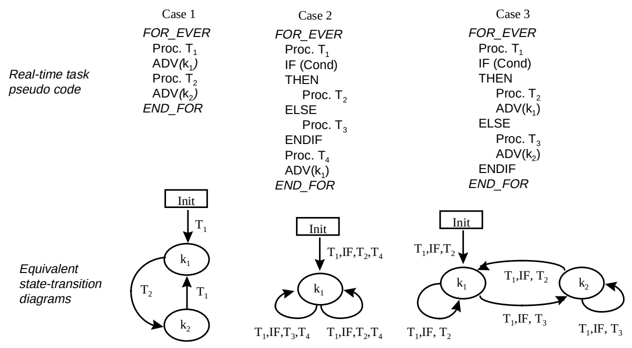
Definition of task state-transition diagrams. An *ADV* instruction splits the task code in two parts: the piece of code before and the piece of code after the *ADV* instruction. It has also a precise semantic: the code « before » must end before the instant defined by the *ADV* instruction and the code « after » will only begin after this instant. So, we define a processing as a piece of code included between two *ADV* instructions, or two *ADV* occurrences if there is only one *ADV* instruction in the code else if the *ADV* instruction is not located at the end of the task main « for ever » loop (for more details, see Figure 2).

A task can now be viewed as series of processings that have precedence relationships. We define a state of the task as the formal instant where it is. So, with each *ADV* instruction in the task code is associated a state of the task. Transitions between those states are done by executing the code included between the two corresponding *ADV* instructions, i.e. the processing.

We use a formalism based on graph theory to represent a state-transition diagram for each task. At each state, we associate a vertex. At each transition, we associate an oriented edge. Since our graphs have only oriented edges, we will further use the word edge to designate an oriented edge. We finally obtain a graph where each vertex is an observable state of the task (corresponding to one *ADV*) and each edge is the processing executed between two *ADV* instructions. Hence , an « *ADV(k)* » state features the deadline of all edges ending to this vertex. Let $G_\omega = (X_\omega , U_\omega)$ be the notation of this graph, X_ω be its set of vertices and U_ω be its set of edges.

When there is a conditional instruction with an *ADV* instruction in the « then » or the « else » body, there are two exit edges at the vertex. We only consider the IF...THEN...ELSE...ENDIF case, but we can extend this to other conditional instructions. We also illustrate the different cases in the Figure 2. The presence of an *ADV* instruction in a conditional statement make the state-transition diagrams more complex: it introduces a new vertex and can greatly increase the number of edges in the graph. It allows to describe more precisely the timing behavior of a task and to make easier the identification of possible conception mistakes on the timing constraints. When there is no *ADV* instruction in any body, there is only one edge for each branch.

When a task begins its execution, we suppose there are two steps, the boot step and the initialization step. The boot step is complex and executed before the task code. We do not consider the boot phase in this paper and will suppose that there is no processing in this phase. But, to take into account the shifting factor of a task (see 2.1), the state-transition diagram is extended by adding a new specific vertex 'Init', labeled by the value of the shifting factor. The initialization step then takes place between the vertex 'Init' and defines its exit vertex(ices). In most cases, it corresponds to the first processing of the task in its code. For example, in the second and third case of the Figure 2, during the initialization phase, we supposed that the initial value of 'Cond' is true.



```
FOR_EVER
  Proc. T1
  ADV(k1)
  Proc. T2
  ADV(k2)
END_FOR
```

```
FOR_EVER
  Proc. T1
  IF (Cond)
  THEN
      Proc. T2
  ELSE
      Proc. T3
  ENDIF
  Proc. T4
  ADV(k1)
END_FOR
```

```
FOR_EVER
  Proc. T1
  IF (Cond)
  THEN
      Proc. T2
      ADV(k1)
  ELSE
      Proc. T3
      ADV(k2)
  ENDIF
END_FOR
```

Figure 2. Some state-transition diagrams

Properties of state-transition diagrams. The state-transition graphs (or diagrams) are connected: as real-time tasks have no termination and as there is no infinite loop in a processing, each vertex possesses at least an exit edge connected with the same or another vertex. For the same grounds, these graphs include cycles: each in-degree and out-degree of a vertex is at least equal to one [16]. Indeed, the number of vertices is finite and they have at least one exit edge and one entering edge (except for the Init vertex which has, by definition, no entering edge).

This graph is capable of handling tasks with multiple deadlines. It represents the symbolic execution of a task, i.e. all the possible execution paths of a task as if it was separately executed on a processor. By construction, there exists an unique edge for a given processing and an unique vertex for a given *ADV* instruction. So, for a given task there exists one and only one state-transition diagram and there is an equivalence relationship between state-transition diagrams and task codes. From now, we can describe a task by its code or by its equivalent symbolic state-transition diagram.

2.3. A brief example

Let ω_1 and ω_2 be two real-time tasks such as $H_{\omega 1} = H_\Omega$ and $H_{\omega 2} = 3.H_\Omega$ (one period of $H_{\omega 2}$ is equal to three periods of H_Ω). This example is a producer / consumer case: ω_1 sends messages (to ω_2) when its data are ready and ω_2 consumes them if it can. We also suppose that the processings, that depend on the boolean value of « data_ready », have not the same timing constraints. We give their formal code description and their corresponding state-transition diagrams in the Figure 3.

```
ω1 such as Hω1 = HΩ
FOR_EVER
/*  T1 Processings */
IF ( data_ready = true )
THEN
        SEND( ω2 , 2Ω )
        /*  T'1 Processings */
        ADV( 2ω1 )
ELSE
        /*  T''1 Processings */
        ADV( 1ω1 )
END_FOR
```

```
ω2 such as Hω2 = 3*HΩ
FOR_EVER
/* T2 Processings */
IF( message_received)
THEN
        /* Message Consumption */
END_IF
ADV( 1ω2 )
END_FOR
```

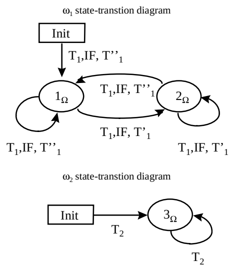


Figure 3. Description of ω_1 and ω_2 tasks

In this example, we use the notation k_ω to design k periods of H_ω. We also use the notation $SEND(\omega, \delta)$ where ω is the message recipient. With this notation, the message will be visible δ instants after its current instant. We will further consider that processing T_2 is also including the test and consumption of messages. Each processing fulfills the assumptions made in 2.1. In the state-transition diagrams, vertices are labeled by the *ADV* value in H_Ω units and edges are labeled by the code of the processings.

We should finally notice that the worst case execution time (WCET) of these processings can be calculated with existing results [17]. Then, if we assign calculated upper bounds to the edges, we obtain a graph were each edge is the processing duration and each vertex represents the timing constraints, i.e. deadlines, on these processings. We will further use this graph to determine the set of linear constraints to guarantee the timeliness in the execution phase.

3. Dependability in safety critical systems

The safety critical systems are complex. Their complexity can be evaluated in a qualitative (e.g. kinds of function) or in a quantitative (e.g. number of code lines) manner. If programmers are not aware of it, the more complex the application and its implementation are, the more difficult the verification of dependability properties is. Dependability is a significant feature that must be considered as soon as possible, in order to be able to verify it. In the dependability properties, we are mainly interested in this paper by three of them: safety, liveliness and timeliness properties. In this section, we first introduce the definition of these three properties and then we give for each property the way they are guaranteed.

3.1. Safety

This property is verified when we are able to demonstrate that some non-authorized system malfunctions (mainly connected with parallelism in our approach) are impossible. One of the most significant characteristic of the safety is the data coherence. There are two kinds of interaction: the temporal variables and the messages. For the first communication class, the new value is broadcasted only after the latest end date of the producing processing. A processing can only read the temporal variable values after its earliest start date. Since the architecture is time-triggered, incoherence cannot happen except if there are failures. For the same reasons and because of the existence of a total order in a queue, the message passing mechanism also preserves the data coherence. Moreover, message queue and temporal variable memory requirements can be sized off-line and no memory saturation can occur.

Another significant characteristic of safety is the verification of application behaviors. In this case, we need to observe the output data, i.e. values and dates of the results, face to input scenarios. The target computer independence, the determinism and the predictability property allow us to make the verification through tools, that are defined in the OASIS approach, such as the actual execution in simulated-time (for more details, see [18]).

The latest characteristic of the safety property we can mention here, is the fault checking and confining techniques that can be implemented. In the application domain we consider that each processing has a bounded duration and that their upper bounds can be exhibited. When executing the tasks, it is possible to verify that this allowed processing time is not exhausted. In this case, we can then define in which conditions the faulty task may be stopped (e.g. duplicating the latest correct values of its temporal variables). If it is effectively stopped, it is done without violating the model (i.e. no timing fault propagation).

3.2 Liveliness

In our context, a system is alive when all critical tasks (i.e. all tasks in the OASIS approach) can always become ready. In the OASIS approach, liveliness is an intrinsic property. Indeed, the only blocking instruction is the *ADV* instruction. When the processing preceding the *ADV* instruction ends, the task must wait its next activation instant defined by the *ADV* instruction before beginning a new processing. The time-triggered architecture ensures that the task will not be blocked and the next processing will also be executed. Liveliness property demonstration is then included in the timeliness property demonstration.

3.3. Timeliness

This property is verified if and only if all the critical task processings are always executed in a timely manner. We also name this property the temporal correctness. This property always true in the design phase has to be demonstrated when we are executing the application. This verification is directly connected with the target computer sizing problem. If the processor is not powerful enough, some tasks could miss their processing deadlines. In the application domain, such situations are considered as major failures and must be avoided. All tasks are critical and have then absolute strict deadlines.

In the OASIS approach, all interactions are time-driven and *all timing constraints* are clearly expressed in the design phase (see 2.2). From these constraints, for each task, all processing deadlines are calculated on the same global real-time clock. Whatever the scheduling policy is, outputs (values and production dates) of the real-time tasks are always the same.

The scheduling policies deal only with reaching the right sizing of the processing units (this is a major concern of our approach). If all calculated deadlines were regular, rate-monotic [19] or deadline-monotic policies could be used in our approach. Nevertheless, as one of the highlights of the OASIS model is its capability of handling tasks with multiple deadlines, we prefer to stay in a generic case. By the way, we can use the results of the deadline-driven scheduling techniques [20]. These techniques are efficient, rigorous and optimal if the programming model is adequate. The OASIS model fulfills these assumptions. We only need to calculate processor load and to make an off-line analytic verification of linear equations and inequalities. It allows us to calculate necessary and sufficient conditions to analyze the schedulability of the tasks and to guarantee that on-line task scheduling still ensures the timeliness property.

Timeliness is also significant as a performance measure for real-time systems [21]. When evaluating performances of real-time systems, a binary qualitative criterion is the timeliness satisfaction, the ability to meet all deadlines. In our approach, we share the same point of view and the evaluation of this criterion can be directly done by the verification of the necessary and sufficient conditions.

4. The timeliness property: the synchronized product

4.1. Problem statement

We saw in section 3.3 that the problem of ensuring timeliness is connected with computer sizing. If we know the basic features of the target computer (CPU speed, etc.), it is possible to size the resources that are necessary to correctly execute an application. Our approach is based on an off-line analysis of all interactions between all the tasks, to guarantee that the on-line scheduling ensures timeliness. We suppose that we use a deadline-driven scheduling policy. If we are in the single processor case, we are interested in knowing if the processor is best-suited, e.g. if its utilization factor is near and less than one.

In the multiprocessor case, the problem is to determine the number of processors and their interconnection topology. In the case the processors have their own memory, communications are realized through the interconnection links, we have to ensure that their timeliness property is also guaranteed. It raises two main problems. First, before sending the message, it is stored in a message queue. The storage date is significant, because it may add timing constraints on processings that are executed before. These timing constraints have to be fulfilled. Secondly, many tasks can send messages in the interconnection network. A real-time communication protocol has to be defined. As for the tasks, its properties should allow us to ensure timeliness in the same way. Although they are significant problems in the multiprocessor case, we will further only consider the single processor case or tasks without communications in the multiprocessor case.

To verify if a task can be executed lonely on a processor, we can define the processor load it generates on a given processor. A real-time task ω_i is constituted by series of processings we identify in its state-transition diagram. For each processing *j*, we know its execution time upper bound (on a given processor) τ_{ij} and its deadline δ_{ij}. From now on, processing execution time will be the upper bound of the processing execution time, i.e. its WCET. We first define the processing load as the ratio between its execution time and its deadline: $\rho_{ij} = \tau_{ij} / \delta_{ij}$. For a task ω_i, we can then define its load ρ_i, as the maximum of its processing loads: $\rho_i = \max_j(\rho_{ij})$. Finally, the inequality $\rho_i < 1$ is fulfilled if and only if the task ω_i can be executed on the given processor.

An application is composed by a set of tasks $\Omega = \{\omega_i \,/\, i \in I\}$. With previous results, we have a sufficient and necessary condition to execute one task on a given processor. We could extend these results to the set of tasks Ω. We then define a set of necessary conditions for each task in Ω, that we name elementary necessary conditions:

$$\forall\, i \in I,\ ENC_i : \rho_i < 1 \quad (1)$$

In our study case, as the communication mechanisms are implemented in an asynchronous way, we can consider tasks as independent. We can then define a sufficient condition for the correct execution of the set of tasks Ω, based on the deadline-driven scheduling analysis.

$$\text{Sufficient condition [20]: } \sum_{i \in I} \rho_i < 1 \quad (2)$$

If we only use this sufficient condition, some correct cases could be ignored or conversely, it could lead to a great oversizing of the target computer. Indeed, this sufficient condition (2) does not take into consideration the behaviors of the tasks: all tasks are considered as totally asynchronous. In real-time systems, synchronization points in time are numerous ; timing task behaviors may not be ignored. In our approach, we can have more accurate results than with the common sufficient condition. Our solution, well-suited for the OASIS approach, is based on the following theorem, which is an refinement of the sufficient condition (2):

> ***Theorem***: At each TT interval of the system, the sum of every pieces of execution times of all active task processings (i.e. executed in the TT time interval) must be less than the interval duration.

This theorem is a necessary and sufficient condition for the OASIS model, because (2) becomes a necessary and sufficient condition if at any time we replace the general factor load ρ by the actual load rest. The problem is then to determine this actual load rest.

The task behaviors are exhibited in state-transition diagrams. When tasks are executed on the same processor, the analysis of their behaviors can be improved if we define a specific operator on their state-transition diagrams to model their mutual arrangements and to be able to apply our theorem. We then introduce in the following sections the results of our works, i.e. the definition and the construction of the synchronized product operator.

4.2. Definition and construction

Definition. We consider a set of N tasks $\Omega = \{\omega_i \,/\, i \in [1..N]\}$ and the set of their associated state-transition diagrams $G = \{G_i=(X_i , U_i) \,/\, i \in [1..N]\}$. The mathematical definition of the synchronized product is an operator from the state-transition graph space (which is a subspace of the square NxN matrix space) to the general matrix space. This operator associates to a vector of N state-transition diagrams a graph that features the simultaneous execution of the N tasks (we will later see the properties of the synchronized product graph). This graph has not the same semantic than the state-transition diagrams, but a vertex will represent the TT time interval shared by the tasks and defined by their states and its preceding edges will represent the processing pieces that can be executed during this TT time interval.

The basic idea behind the synchronized product is not new, but was not yet considered in our way. For example, composition of synchronous automata was introduced in the « synchronous approach » and merging of asynchronous automata was proposed for languages like Occam. The synchronized product we use does not produce a graph that is equivalent to the task execution, but a graph that is useful for task execution analyses.

The construction technique. To illustrate the technique we use to construct a synchronized product, we first describe a small example with two tasks in Figure 4.

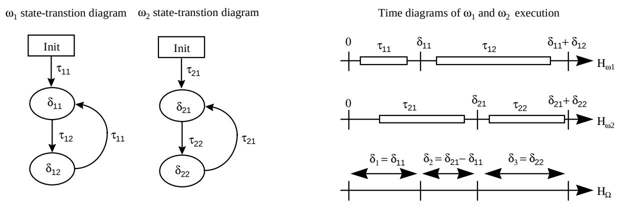
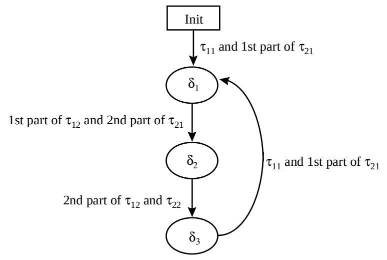


Figure 4. An example with two tasks

In this example, the simultaneous execution of the two tasks is a state-transition graph with three nodes. Because they are executed on two TT time intervals, the first processing of ω_2 and the second processing of ω_1 are split in two parts. The graph of the simultaneous execution of the two tasks is presented in Figure 5.



Init
τ_{11} and 1st part of τ_{21}
δ_1
1st part of τ_{12} and 2nd part of τ_{21}
δ_2
τ_{11} and 1st part of τ_{21}
2nd part of τ_{12} and τ_{22}
δ_3

Figure 5. Simultaneous execution of ω_1 and ω_2

The construction of the synchronized product of N tasks is based on the generalization of the technique previously used. We use the matrix algebra and the equivalence between a graph and its incidence matrix to explain the construction of the synchronized product operator.

First, we search the smallest period, namely *ref*, shared by all the tasks (the « worst case » happens when it is equal to the H_Ω period). The values of each vertex of each state-transition diagram in G are translated in *ref* units. Formally, a state or vertex $X_{i,j}$ of a graph G_i can now be split in series of $n_{i,j}$ ($n_{i,j} = v(X_{i,j}) / ref$) successive substates that we note $\{X_{i,j}^k / k \in [1..\ n_{i,j}]\}$ (we use the notation $v(u)$ for the value of the vertex u). We now consider N-dimension vectors composed by the combination of each task substate with the other task substates:

$V_k = (X_{1,j1}^{k1} ;\ \ X_{2,j2}^{k2} ; \dots ; X_{N,jN}^{kN})$ *with* $\forall p \in [1..N]$, $k_p \in [1..\ n_{i,jp}]$ *and* $X_{p,jp}$ *is a substate of graph* G_p

Because of the state and substate precedence relationships, only few of these states really exist. Indeed, between two substates of a task there are two kinds of precedence relationship, those due to the substate decomposition and those due to the task behavior represented in the state-transition diagram.

Now, we construct the matrix that describes the precedence relationships between all these N-dimension vectors: an element in this matrix means that one vector is followed by another one and gives the features of the connecting edge(s). In the other case, we put a null-value in the matrix. One vector is followed by a second one if and only if each substate of the first vector is followed by each substate of the second vector. The resulting matrix is in fact the incidence matrix of the synchronized product graph of all the tasks. Finally, we associate to this matrix its equivalent graph representation. We now designate the synchronized product operator by the 'Π-operator' notation and the graph of the synchronized product by the 'SP-graph' notation.

For the initial vertex of all task state-transition diagrams, they will constitute an unique vertex that will be the 'Init' vertex of the SP-graph.

4.3. Properties of the SP-graph

The SP-graph represents all the possible arrangements of all the processings of all the tasks that are assigned to one processor. Moreover, this graph is consistent with the execution of all the tasks: a path in this graph represents a possible execution path of the given tasks. Because of the construction technique, the uniqueness of the state-transition diagrams and the predictability property of the TT architecture, this graph will be unique. Finally, this graph will have the same properties than the state-transition diagrams: the graph is connected and each vertex has at least one exit edge and one entering edge (except for the initial vertex), it includes cycles.

4.4. Linear system generation

The generation technique. In the SP-graph, we saw the formal definition of each vertex and each edge. This definition is connected with the symbolic view of the state-transition diagrams. A vertex in the SP-graph has necessary an entering edge (except for the 'Init' vertex). This vertex then represents the amount of time that is allowed to the processings that are associated to its entering edge. So, the entering edge represents the set of N pieces of the associated processing execution times. Consequently, these execution time pieces are verifying the property that their sum is less than the time assigned to the vertex value. We designate these execution time pieces with free real variables included in the [0,1] interval. Then, in the SP-graph each edge will represent the piece of execution time that will be done during the time interval of the vertex that follows the edge.

We can identify in the SP-graph the series of substates that compound a task state. All time execution pieces of the edges connecting these substates compound the unique processing. The sum of all these execution time pieces must be equal to the total processing duration, which is the value of the corresponding transition in the state-transition diagram. Consequently, the sum of the free real variables associated to those execution time pieces is equal to one.

Finally, we obtain a set of inequalities generated at each edge of the SP-graph. We only need to cover all the edges of the graph to exhibit the linear inequalities with the free variables. Due to the state splitting, we obtain linear equations on the free variables (that compound the task processings). Hence, examination of all possible processing splitting in the SP-graph suffices to generate these equations. We have to complement this linear system with inequalities due to the domain definition of the free variables.

For the small example presented in the Figure 4 and the Figure 5, we will have the following linear system:

$$\begin{array}{ll} \exists\, \alpha \in [0,1] & (\tau_{11} + \alpha.\tau_{21}) / \delta_1 < 1 \\ \exists\, \beta \in [0,1] & (\beta.\tau_{12} + (1-\alpha).\tau_{21}) / \delta_2 < 1 \\ & ((1-\beta).\tau_{12} + \tau_{22}) / \delta_3 < 1 \end{array}$$

Only α and β are the free variables of the system. If this system of linear constraints has a solution, the two tasks can be correctly executed on the same processor, i.e. the timeliness property will be guaranteed.

On the one hand, to demonstrate that timeliness is ensured for a set of tasks executed on a given processor, the values of the free variables of the linear system we obtain are not interesting in our approach. The property of the system that only interests us is the one of having a solution for some values of the free variables in their definition domains. Indeed, if the linear system has a solution, the time-driven scheduling can be applied and is optimal. So, when the application is executed, the values of the free variables are useless.

On the other hand, if we replace the right member of the inequalities by a free variable *k* (*k* in the real domain), and we search the minimum value of *k* for which the system still has a solution, we will obtain a maximum task load generated by the tasks if they are executed on the same processor. More precisely, this value represents the necessary ratio of time that will be used to execute the tasks at each TT interval.

The timing constraints due to the send of messages can be taken into account in our work as it will be shown in section 5.

The solving technique. We first need to make the inequalities become homogeneous. For this reason, we divide all the inequalities by their right member. We still have a linear system with real variables and factors. The right member is then the unit. We replace the right member by a free real variable k_0. For some values of k_0, the system has or does not have solutions. We search the minimum value of k_0 for which the system has still a solution. It will in fact represent the maximum load, i.e. the true load, of the tasks on the processor. We use for this value the notation $(k_0)_{max}$. If $(k_0)_{max}$ is less than the unit, the set of tasks can be correctly executed (i.e. enforces the timeliness property). In the other case, the set of tasks cannot be executed on the chosen processor and we must change at least one system parameter (the processor performances, a timing constraint or the task decomposition in the application). An application of this technique is the sizing of the target computer. Indeed, we notice that we could also apply this technique to search the minimum number of processors necessary to correctly execute a given application on a multiprocessor computer.

The linear system we obtain is a common set from linear algebra with real variables and the minimum search is a classical problem. We can formulate our problem as following: let A be the matrix of inequalities, B be the matrix of equations, $\underline{X}$ be the real free variables vector and $\underline{0}$ (or $\underline{1}$) be the vector full of 0 (or full of 1):

$$\text{Min } [f(\underline{X}) = k]$$
$$\text{with } A.\underline{X}-k.\underline{1} \le \underline{0} \text{ and } B.X = \underline{1}$$
$$\text{where } \underline{0} \le \underline{X} \le \underline{1}$$

We can use the results on the Constraint Satisfaction Problems, i.e. mathematical programming techniques to solve this problem [16]. As we only work with numerical real numbers, the search of a solution is easier if we use techniques like linear approximations.

If we are not aware of it, this technique can generate in some cases, e.g. for tasks with great differences in their timing parameters, a large linear system in number of inequalities and free variables. Nevertheless, it is possible to reduce the complexity of the system, by reducing the SP-graph (see section 5.3). Such a technique will be based on the identification and elimination of redundant inequalities (same terms and variables) and useless terms (e.g. when there is no processing).

5. A case study

5.1. Presentation of the example

Let the operational application be a software management of a safety device. We suppose that the system supervises, displays data and alarms connected with the state of the core in a nuclear power plant. Its mission is to prevent the risk of appearance of bubbles of steam in Water Pressurized Reactor. For safety grounds, we suppose there exist two identical systems which are not synchronized. They exchange their data taken from the environment, and their results from their own computations, to ensure coherent results and to identify failures. Each of it first takes the data from the environment and compares a first group of results. While it continues its calculations with the last received results, it sends its own results to the other one. Finally, it displays the possible alarms if an anomaly is detected.

We introduce a printer to edit historical traces, and an operator which may act on supervision parameters, and may choose mode traces. We suppose that there exist two main communication links. The first link is used to exchange the data with the other system, the second link is used to communicate with the operator.

The presence of a human operator during a computing cycle makes the management of this system more intricate. It must be correctly taken into account. OASIS must do a safe gathering between the elements which communicate but which also evolve in an asynchronous way. The global functioning is based on a main periodic set of tasks with a period of Δ_{cycle} units of time (or *ut*).

5.2. Description of the tasks

The application is compounded of five different sets of tasks, we detail in this part. The first set of tasks is the operator management task and the receiving task, which are totally asynchronous tasks. We will not consider them in the synchronized product, because they can be modeled as a constant load value for each TT-interval.

The second set of tasks is reading sensor data and all the features of the environment (thermocouples, state of the pumps, pressures, etc.). Data are collected during the first part of the main cycle and before the beginning of calculations on them (during Δ_{read} ut). We suppose that the data sensors are controlled by the system. We suppose sensor data are read n_{read} times (every δ_{read} ut). There are as many tasks as sensor numbers (i.e. n_{sensor} tasks).

There is also a computational task that provides the calculations based on the sensor data stream to determine the state of the core it supervises. This task has two processings that must be done in $\Delta_{compute}$ ut in a main cycle. Computation1 and computation2 are the two consecutive processings in the computation task. This cut off comes from a timing constraint due to a send during the computation.

The computational task has to send its first group of results to the other safety device, so there is a task to send data on the communication link. This « sending » task has to begin its processings D_{send} ut after the beginning of the cycle (see Figure 6). Data are split in n_{send} groups and are sent every δ_{send} ut.

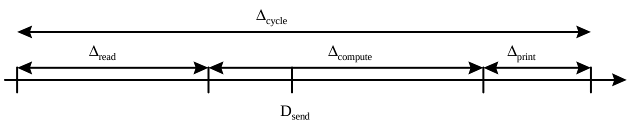
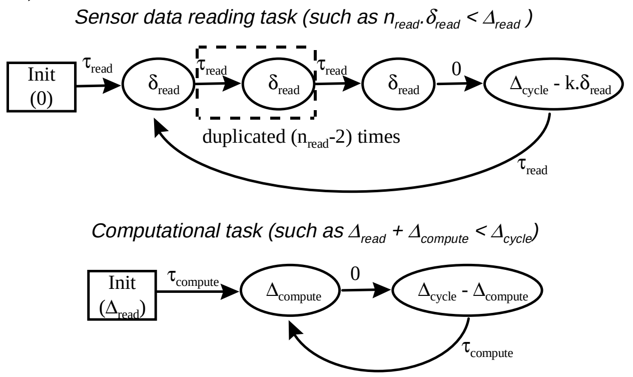


Figure 6: Timing constraints

Finally, there are tasks which aim to display the alarms and print the data. The first task has a deadline of Δ_{alarm} ut and the second task has to print data in n_{print} steps, every δ_{print} ut (during a maximum duration of Δ_{print} ut).



Sensor data reading task (such as $n_{read}.\delta_{read} < \Delta_{read}$)
Init (0)
τ_{read}
δ_{read}
τ_{read}
δ_{read}
τ_{read}
δ_{read}
0
Δ_{cycle} - k.δ_{read}
duplicated (n_{read}-2) times
τ_{read}
Computational task (such as $\Delta_{read} + \Delta_{compute} < \Delta_{cycle}$)
Init (Δ_{read})
$\tau_{compute}$
$\Delta_{compute}$
0
Δ_{cycle} - $\Delta_{compute}$
$\tau_{compute}$

Figure 7. Simplified state-transition diagrams

To describe the state-transition of the real-time tasks, we use the notations introduced in section 2.2. The two main types of these graphs are illustrated in the Figure 7. A zero value means there is no processing to compute. The « send » task, the « print » task have the same graph than the sensor data reading task, if we replace the shifting, *ADV* and processing values by their respective values. The « alarm » task has also the same graph than the computational task.

5.3. Experimental results

To obtain experimental results on this case study, we assigned real values to the different timing parameters of the example presented above (see Figure 8).

H_Ω period = 10 ms	Δ_{read} = 1s	τ_{alarm} = 20 ms
δ_{read} = 50 ms	Δ_{send} = 500 ms	τ_{print} = 20 ms
δ_{send} = 100 ms	Δ_{print} = 1s	n_{read} = 10
δ_{print} = 100 ms	D_{send} = 2 s	n_{send} = 5
Δ_{alarm} = 500 ms	τ_{read} = 10 ms	n_{print} = 10
$\Delta_{compute}$ = 3 s	$\tau_{compute}$ = 200 ms	n_{sensor} = 10
Δ_{cycle} = 5s	τ_{send} = 10 ms	

Figure 8. Timing parameters

The computational task sends a message to the sending task. The deadline of the message is D_{send}. So, if we want to take message sending into account, we need to add one inequality in the system: the sum of all pieces of load of $\tau_{compute}$ on the TT-interval before D_{send} is greater than or equal to $\tau_{computation1}$.

The final synchronized product of these tasks has 901 vertices and 901 edges. The corresponding linear system (without reduction) has then 901 (number of edges) inequalities, 56 equations on 4500 free variables. The solving technique takes 28 seconds on a Sun Ultra-1 station and gives a maximum load of 0.684 (with a 10^{-3} precision). The application can then be executed on the given processor and its timeliness is ensured.

6. Conclusions and future works

The works presented here extracted from a more important project, the OASIS project. Works on the OASIS project have a more global span. Issues from the design to the execution of the safety critical systems are the main focus to the works in the OASIS project. Its technical objectives are to propose a complete chain of development: compilation & simulation tools, safety embedded runtime & microkernel are under development. These former works deal with fault-tolerant problems and techniques (such as fault-checking and fault-confining techniques) in order to ensure the required level of reliability and safety.

We presented a systematic approach to verify the timeliness property of real-time safety critical systems. The timeliness verification is mainly based on the theorem presented in section 4.1. We also presented the way to realize the verification of timeliness, with our real-time task model, the state-transition diagrams. This verification is done by the definition and the construction of the Π-operator, that composes the symbolic diagrams of the real-time tasks and generate the graph of all the possible arrangements among them. We use this graph to generate the linear system. With the case study, we show that our approach can handle realistic problem sets and its effects on moderate sized problems. There is no gap between our model, its implementation and its verification, because we are able to incorporate all the microkernel overheads to the WCET of the tasks. The SP-graph construction tool has been developed: it implements the Π-operator, generates the complete linear systems, and calculates a maximum load of the input set of tasks, described trough their state-transition graphs.

The main benefits of our approach deal with the dependability and parallelism adequacy issue and the computer sizing problem. We saw that timeliness is a significant part of performance evaluation for safety critical systems. In this context, the computer sizing problem is significant, and the timeliness verification is strongly connected with the classical mapping problem. We are now working on the communication problem. When we apply the same technique, we obtain non linear systems and we are currently working on solving techniques of such systems. Other extensions of our works deal with analyses and interpretations that may be done on the symbolic SP-graph. For example, we could use it to assist the design and the implementation of safety critical applications, by analyzing more precisely the properties of the SP-graph (by exhibiting the time critical execution path in the SP-graph, etc.).

Acknowledgements

The OASIS project is led by the CEA, the french atomic agency. The industrial partners of this project are EDF (french nuclear power plant operating company) and Framatome (french nuclear power plant manufacturer).

Works on the multiprocessor case are made in collaboration with the research team of Pr. T. Muntean from the LIM laboratory (computer science laboratory of Marseilles - France). Works on the actual execution tool in simulated-time are made with the research team of Pr. J.-J. Schwarz from the INSA research organism in Lyon (France). We would also like to thank the other members of the OASIS team: J. Delcoigne and M. Aji.

References

[1] J. A. Stankovic, “Misconceptions about real-time : a serious problem for next-generation systems,” IEEE Computer, vol. 21, pp. 10-19, 1988.

[2] P. R. H. Place and K. C. Kang, “Safety-critical software : status report and annoted bibliography,” Technical Report CMU/SEI-92-TR-5 (ESC-TR-93-182), June 1993.

[3] J. A. Stankovic and K. Ramamritham, “What is predictability for real-time systems ?,” Real-Time Systems, vol. 2, pp. 247-254, 1990.

[4] G. Berry and G. Gonthier, “The synchronous programming language ESTEREL, design, semantics, implementation",” INRIA, Research Report-842 1988.

[5] C. A. R. Hoare, Communicating Sequential Processes: Prentice-Hall International, 1985.

[6] M. L. Dertouzos and A. K.-L. Mok, “Multiprocessor on-line scheduling of hard-real-time tasks,” IEEE Transactions on Software Engineering, vol. SE-15, pp. 1497-1506, 1989.

[7] K. Ramamritham, “Allocation and scheduling of complex periodic tasks,” 10th IEEE International Conference on Distributed Computing Systems, 1990.

[8] D. D. Kandlur, K. G. Shin, and D. Ferrari, “Real-time communication in multihop networks,” IEEE Transactions on Parallel and Distributed Systems, vol. 5, pp. 1044-1056, 1994.

[9] T. F. Abdelzaher and K. G. Shin, “Optimal combined task and message scheduling in distributed real-time systems,” 16th IEEE Real-Time Systems Symp., Pisa, Italy, 1995.

[10] H. Kopetz, A. Damm, C. Koza, and M. Mulozzani, “Distributed fault tolerant real-time system: the MARS approach,” IEEE Micro, pp. 25-40, 1989.

[11] V. David, M. Aji, J. Delcoigne, C. Aussaguès, and C. Cordonnier, “Le modèle de conception OASIS/YC pour les systèmes temps-réel complexes critiques,” Real-Time & Embedded Systems Conference, Paris, 1996.

[12] C. Aussaguès, C. Cordonnier, M. Aji, V. David, and J. Delcoigne, “OASIS: a new way to design safety critical applications,” 21st IFAC/IFIP Workshop on Real-Time Programming (WRTP'96), Gramado, Brazil, 1996.

[13] H. Kopetz, “The Time-Triggered approach to real-time systems design,” in Predictability Dependable Computing Systems, ESPRIT - Basic Research Series, B.Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, Eds.: Springer-Verlag, pp. 53-78, 1995.

[14] V. David, “Approche synchrone et objets actifs,” , vol. Receuil des actes des Journées d'Electronique'95 : "Electronique et Informatique pour la sûreté". Centre d'Etudes de Saclay: LETI (CEA-DTA), 1995.

[15] H. R. Callison, “A time-sensitive object model for real-time systems,” ACM Transactions on Software Engineering and Methodology, vol. 4, pp. 287-317, 1995.

[16] M. Gondran and M. Minoux, Graphes et Algorithmes, 3e edition ed: Editions Eyrolles, 1995.

[17] P. P. Puschner and A. V. Schedl, “Computing maximum task execution times - a graph-based approach,” Real-Time Systems, vol. 13, pp. 67-92, 1997.

[18] C. Cordonnier, J. Delcoigne, and J.-J. Schwarz, “Actual execution in simulated-time for assisting the design of safety critical multitasking systems,” Xth IEEE Real Time Conference, Beaune, France, 1997.

[19] M. H. Klein, T. Ralya, B. Polak, R. Obenza, and M. G. Harbour, *A practitioner’s handbook for real-time analysis*: Kluwer Academic Publishers, 1993.

[20] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard real-time environment,” Journal of the ACM, vol. 20, pp. 46-61, 1973.

[21] W. A. Halang, R. Gumzej, and M. Colnaric, “Measuring the performance of real-time systems,” 22nd IFAC/IFIP Workshop on Real-Time Programming (WRTP'97), Lyon, France, 1997.