Analysis of Real-Time Systems Scheduling Using MARTE

GODARD Wenceslas
EADS France
D42 BP90112 31703 BLAGNAC
FRANCE

wenceslas.godard@eads.net

VALENTIN Marie-Line
Airbus
316, route de Bayonne 31060 TOULOUSE
FRANCE

marie-line.valentin@airbus.com

KORTMANN Peter
Tri-Pacific Software Inc.
909 Marina Village Parkway #283Alameda, CA 94501
USA

peter@tripac.com

GERHARDT Mark
Tri-Pacific Software Inc.
909 Marina Village Parkway #283Alameda, CA 94501
USA

mark@tripac.com

ABSTRACT

The "Schedulability Analysis Modelling" (SAM) package of UML 2.0 profile MARTE for Modelling and Analysis of Real-Time Embedded systems is providing support for schedulability analysis with RMA-type techniques. The goal is to help software architecture designers to guarantee that processes always meet their deadlines. We have tried to take the simplest approach as possible to characterize the timing and resource properties necessary to specify a complete set of inputs for a rate monotonic schedulability analysis. Our goal was to take a "usable" approach for the application of the MARTE profile. Our work is based on the OMG standard MARTE, and proposes a subset which can express all the useful and required information for the verification stage. This study is targeting the aeronautical domain where some avionics systems have strong real-time requirements and would benefit from early scheduling analysis. The goal of this paper is therefore to propose a methodology for designing critical systems scheduling and verifying them with RapidRMA™ tool, and to experiment it on a representative avionics application.

1. ANALYSIS OVERVIEW

This paper presents a methodology for real-time software designers to capture and analysis the timing requirements of their architecture. The goal is to ensure the application will execute under worst case conditions using the available resources and within the time allocated to perform the required tasks. The schedulability analysis tool used is RapidRMA™ from Tri-Pacific Software Inc., it provides the capacity to analyze single node, multiple nodes or dependent end-to-end architectures using Rate Monotonic Analysis (RMA), and also provides analysis for cyclic and aperiodic tasks. Note that this software is available as an IBM Rhapsody™ plugin and some figures of this paper are actually snapshots from this tool. RapidRMA™

requires information about the different tasks: this is mainly the period, working time, deadline, priority and the resources used. In practice, these attributes may change over the development process. For example, working time need to be initially estimated and later refined. As a consequence, in order to guarantee a consistency of analysis results from design to implementation stage, our approach aims at being integrated within existing modelling framework. Emerging UML2.0 profile MARTE [1] is by construction tailored for Modelling and Analysis of Real-Time and Embedded systems, it also gives support for schedulability analysis through the SAM package. Our goal is therefore to build on top of this, using an existing modelling framework implementing MARTE.

2. TECHNICAL APPROACH

The Annotation Process and Rationale

Timing properties will be implemented by applying MARTE stereotypes to class instances and message instances on Sequence Diagrams. In the case of shared mutexed resources, stereotypes will be applied to the operations of the class. We have chosen a minimal set of stereotypes that can be used rather than the large number used in the examples available in the OMG UML MARTE specification. Our objective is to create a more usable subset of MARTE concepts dedicated to the annotation of hard deadline performance parameters. This will enhance the adoption of this profile by the UML real-time user community.

This simplified approach was chosen because the details related to the allocation of logical processes to physical entities (with the exception of CPU) are intrusive and not helpful for chedulability modelling. The execution time for a schedulable thread is entered into a RMA model as an aggregated duration. This execution duration characterizes a specific CPU, cache, memory residency, and paging algorithm strategy. The time entered into the analysis model is obtained either by profiling or extrapolation of a known execution time for the specific CPU deployment context. Detailed allocation of memory and description of other CPU related resources only add complexity and are useless for this type of characterization.

The definition of scheduling dependencies between schedulable entities is not explicit within MARTE. Additional attributes have to be added to UML design to facilitate annotation for predictable DPCP (Distributed Priority Ceiling Protocol) and multi-processor deadline and for system timing requirements. The properties contained within each class instance and its relevant events and messages can be used appropriately for processor schedulability analysis, DPCP and aggregate priority ordering in the analysis tools.

Current definition of “resource” in RapidRMA™ is a synchronously accessed mutex with associated timing properties (acquisition time, deacquisition time, operation time). We have adopted the synchronous operation approach rather than the lock-unlock semantics used in the OMG UML MARTE style. We have done this so that the arbitration policy per resource (PCP, Highest Lockers, Basic Inheritance, and No Preemption) is enforceable within a protected operation using synchronous calling conventions.

Approach to handle Communication Needs and Scheduling

The MARTE profile is the result of a thorough job with regard to resource characterization of many types of resources. Network and Communications needs and communications execution resources (busses, message clients, etc.) are characterized in a similar manner to computation needs and computation resources (CPUs, executable task instances, etc.). The symmetric definition of resource types can be observed in Figure 1, from the MARTE profile:

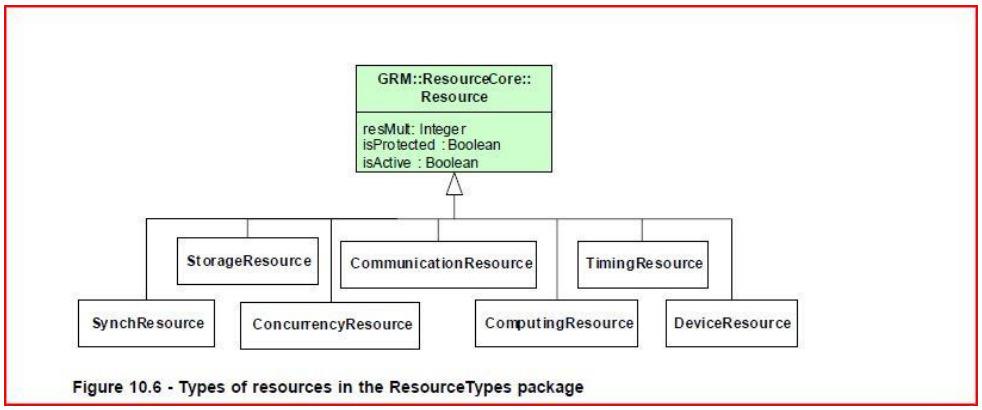

Figure 1 Symmetric Resource Definition for Communication and Computation

As a consequence of the symmetric handling of communication resources and computation resources, a sequence diagram may contain execution instances that use CPU resources or communications resources in a straightforward manner. Consider the following example from the MARTE draft in Figure 2:

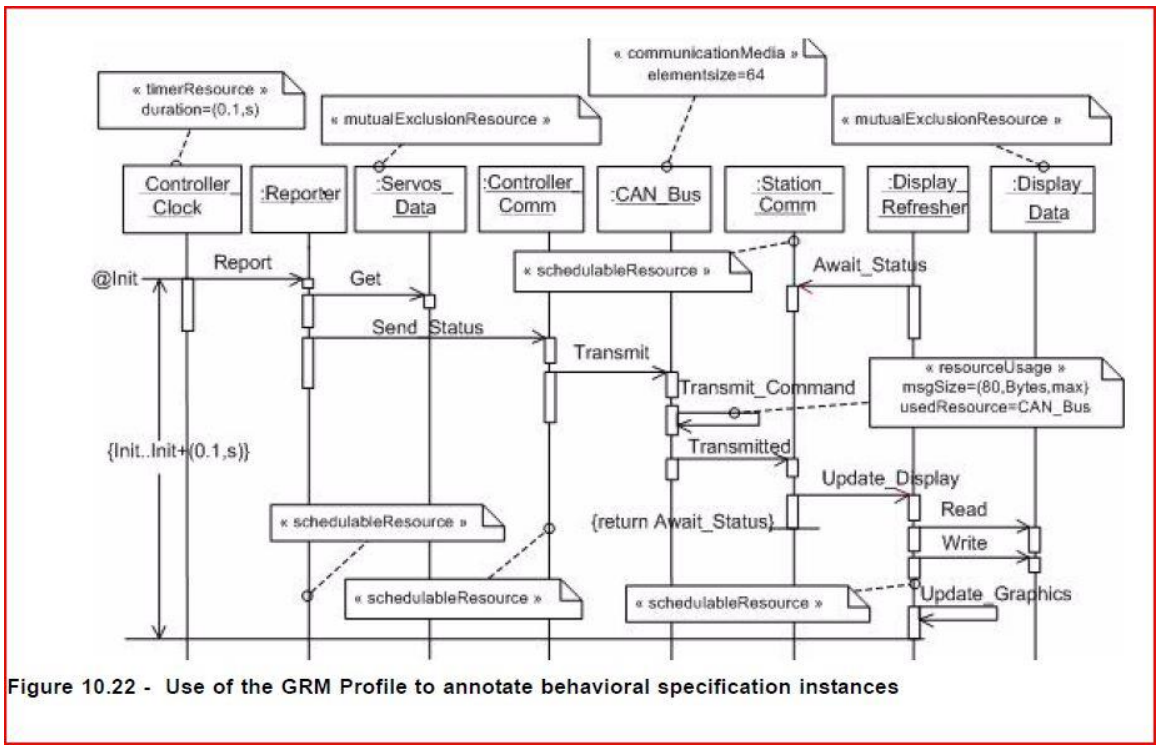

Figure 2 Computations and Messaging in a Sequence Diagram

Note that the Reporter and Controller_Comm columns in the sequence diagram represent executable task instances, while the CAN_Bus column represents a messaging use of the CAN_Bus. The Station_Comm column represents the software that needs to be aware that the CAN_Bus was successfully used. The CAN_Bus column uses network resources while the other columns use CPU resources. Both these uses are directly connected with messaging arrows between their respective portions of the sequence diagram.

Communication has now been factored in on an "equal basis" to computation needs for timing analysis purposes. No further exploitation of attribution is necessary other than to use communications resources and their instances within use case or sequence diagrams/scenarios and intersperse them with the CPU-based

execution instances. Dependencies and causalities are all taken care of by the messaging control protocol expressed in the sequence diagram between source and destinations of the message arrows. Therefore, the network-related stereotypes provided in MARTE combined with judicious use of communication resource execution instances in the sequence diagram provides a complete notation and parametric characterization.

Our design approach through a series of examples

We will present our design approach by describing through examples our recommended usage of MARTE features. This will be followed by a discussion and generalization of what has been presented in the examples.

The first example deals with a representative "straightforward" multi-threaded scheduling problem. It has concurrent threads, resource use, and non-preemptible sections. This one example covers the bulk of the annotation needed for the majority of schedulability analysis problems. This example features a single CPU. The second example shows a multi--CPU architecture and adds the notion of task dependency. The third example shows multi-CPU architecture and the synchronization of a set of terminating tasks.

First example -- Single CPU: threads, resources, non-preemptible regions

A sequence diagram characterizing the system to be analyzed is shown below in Figure 3.

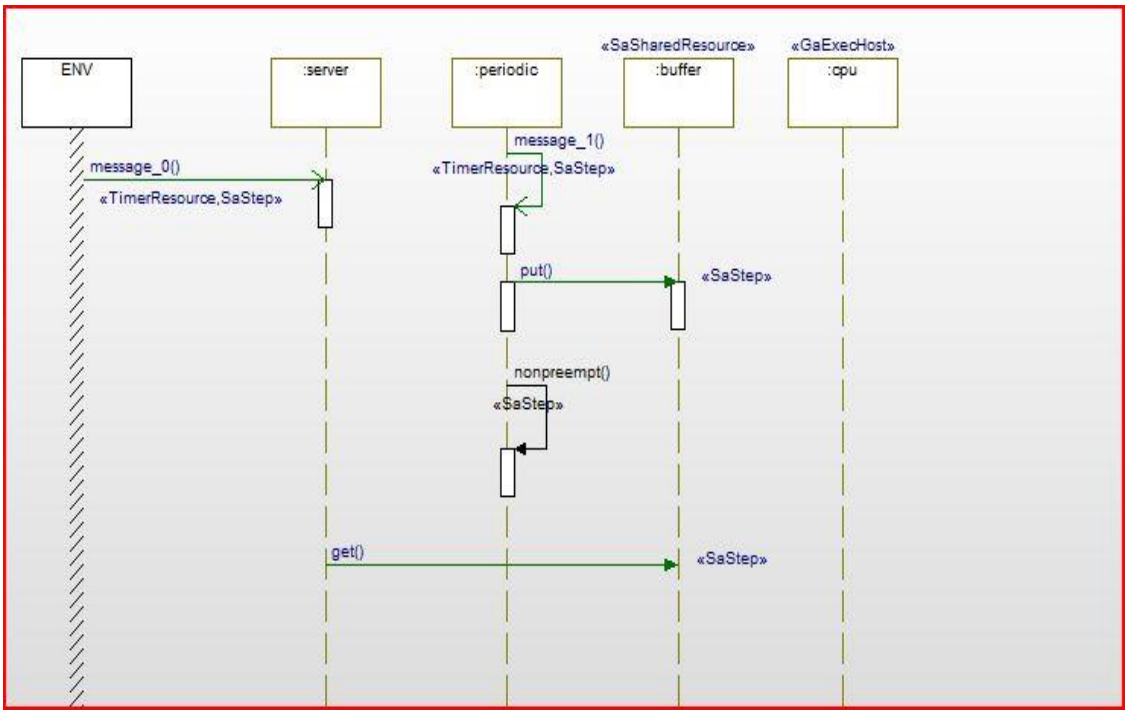

Figure 3 Single CPU Multithreaded Example

There are two periodic threads to be executed: *server* and *periodic*. We have shown two different possible UML styles that can invoke a periodic thread. In our analysis for processing of UML sequence diagrams, open-headed arrows arriving at a class instance on a sequence diagram denote the activation and the invocation of a new executing instance of the class – i.e. a new schedulable element. The instance of server is invoked by an event that generates a message from the UML environment (the column with the striped vertical line) to the class instance named *server*. This is an open-headed arrow, which is realized as an event within UML. This will generate an executing instance of *server* for each message arrival. Note that we have applied the MARTE stereotype for <<TimerResource>> to *message_0* in the UML diagram. We have also

applied the MARTE stereotype for <<SaStep>> to the message as well. The result of the application of these two stereotypes yields the tags for *message_0* in the sequence diagram as shown in Figure 4 and Figure 5 below. Figure 4 shows the tags that result from the application of the <<TimerResource>> stereotype and Figure 5 shows the tags that result from the application of the <<SaStep>> stereotype.

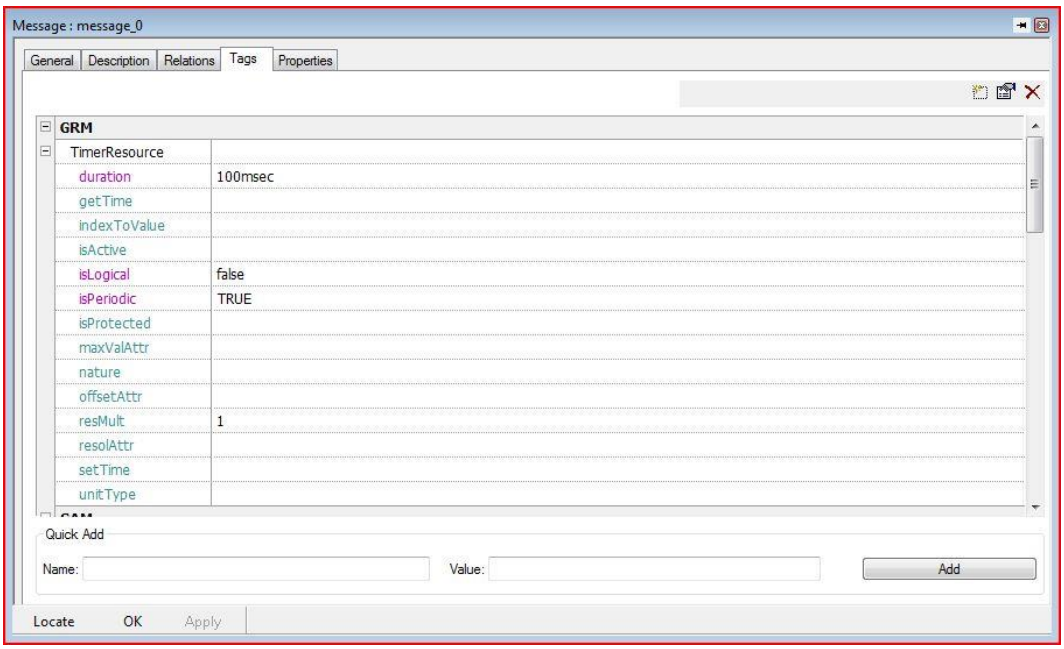

Figure 4 Tags for *message_0* from <<SaStap>> stereotype

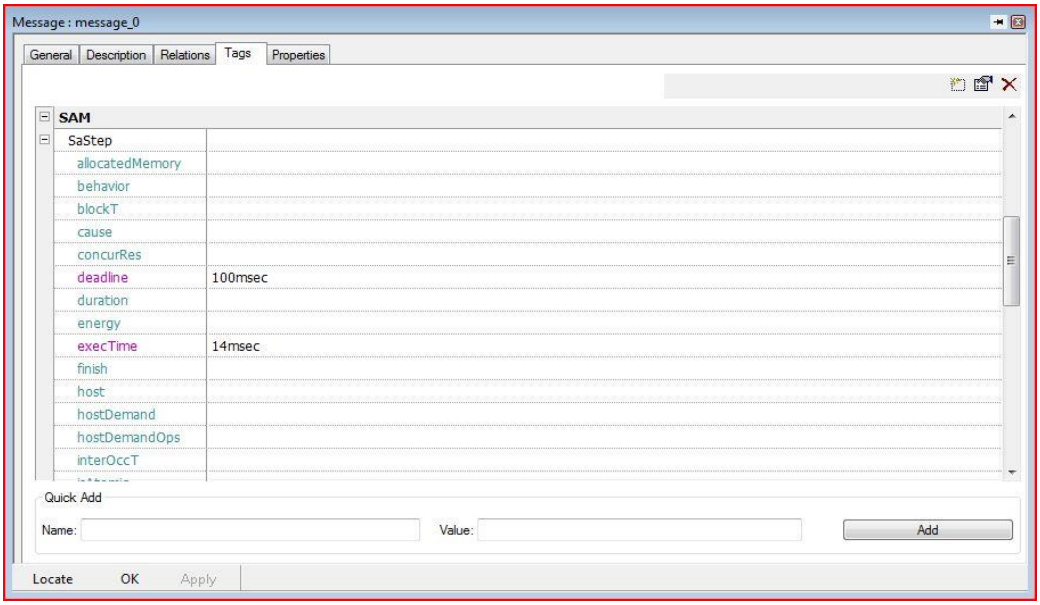

Figure 5 Tags for *message_0* from <<TimerResource>> stereotype

The application of these two stereotypes has allowed us to characterize a periodic occurrence pattern, a deadline, and the amount of execution work to be performed for each executing instance. These are the basic properties necessary to characterize an executing task for RMA.

An execution instance of *server* is created by a message arriving at *server* according to the occurrence pattern as caused by an event in the UML world, which also has the name *message_0*. *Message_0* is caused by the occurrence of an event named *message_0*, which occurs in the UML runtime environment. This is the reason that *message_0* on the sequence chart is realized as an event named *message_0*. See Figure 6 below:

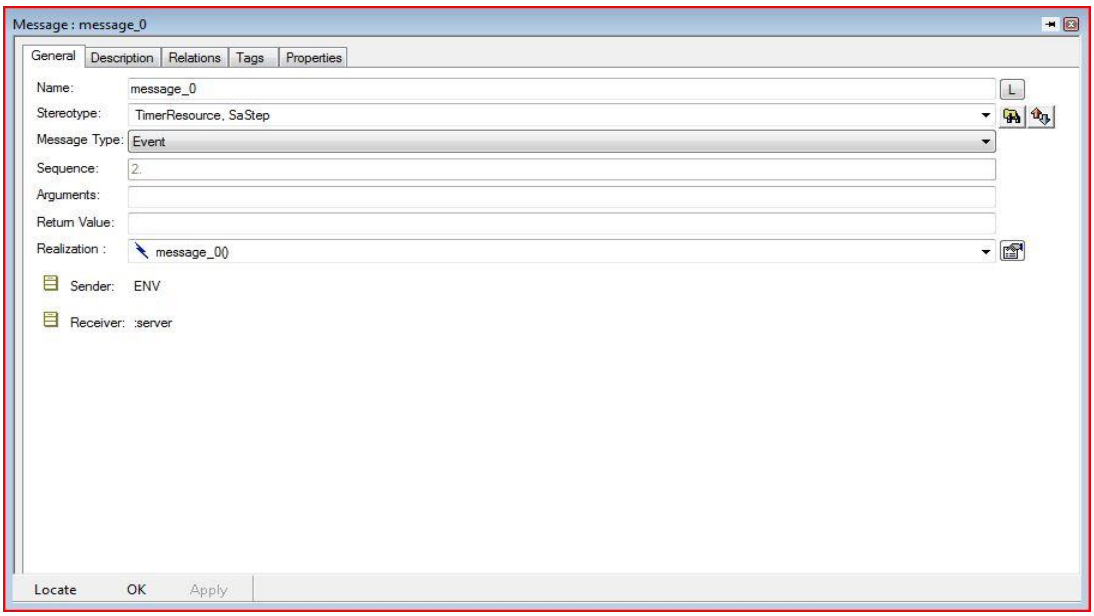

Figure 6 Message Realization as an event

The self directed message labelled *nonpreempt* in the sequence diagram also has the <<SaStep>> stereotype applied to it and has the same properties that have already been discussed for *message_0* as tags. In addition, one other property contained as a tag within <<SaStep>> is used to indicate non-preemption. This property is shown in the screenshot below, which shows additional tags contained within <<SaStep>>.

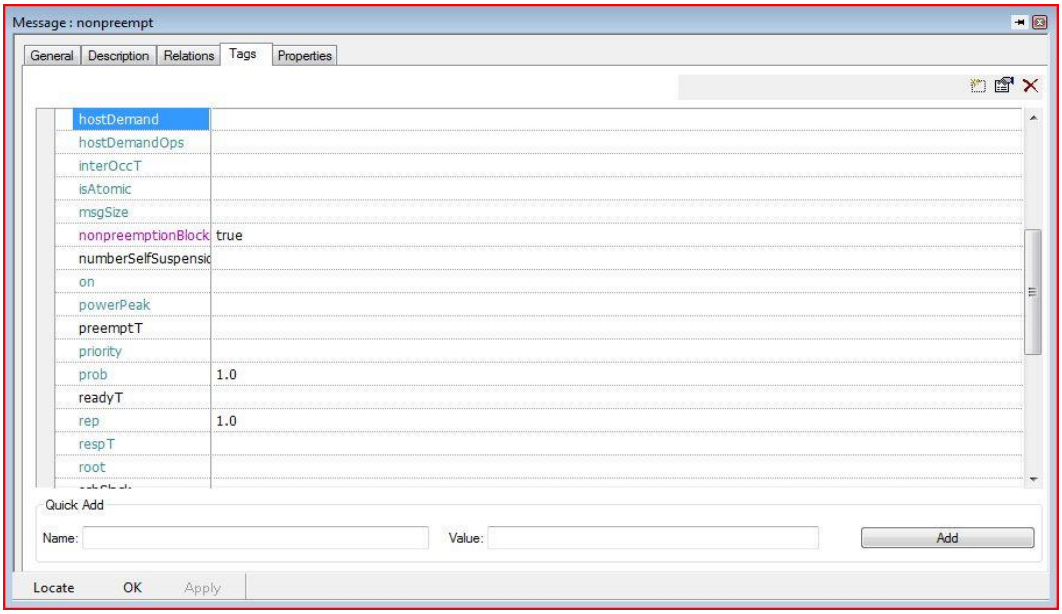

Figure 7 Non-preemption property from <<SaStep>>

The MARTE profile is a highly complex and highly interactive collection of packages, types, tags, and relationships. Addressing the problem of modelling logical threads down to a physical CPU requires the use of a surprisingly large part of the MARTE profile. Rather than going through the complex description of the multilevel allocation structures between a processor and a CPU and a computing step, which are unnecessary for performing RMA, we have used the Host attribute of the <<SaStep>> stereotype.

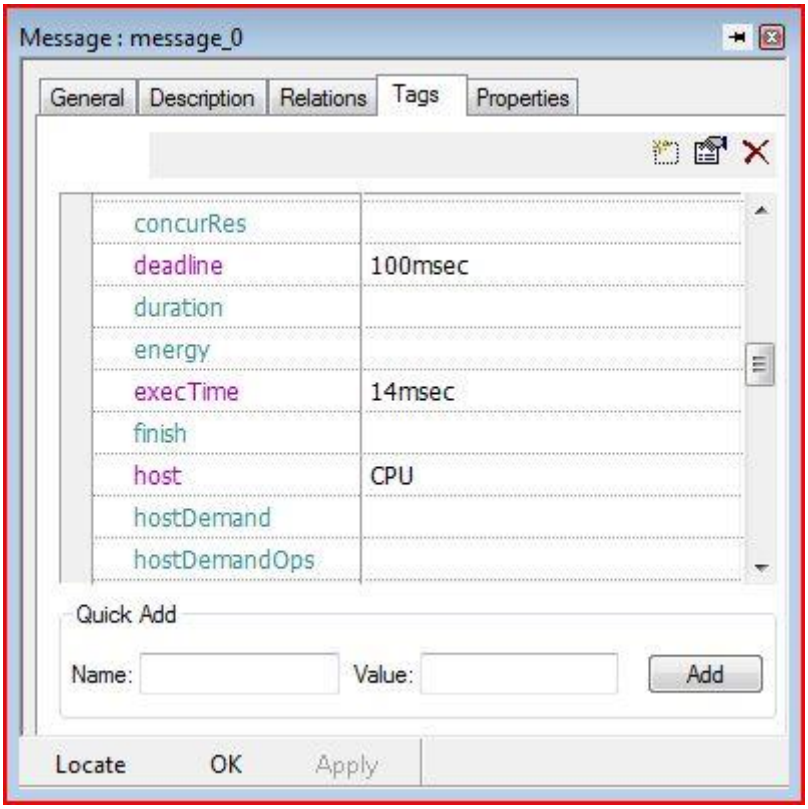

Figure 8 CPU Specified in Host attribute

The CPU Object, which appears in the Sequence Diagram of Figure 3 represents the deployable CPU and has the stereotype <<GaExecHost>> applied to it. This allows assignment of the CPU object to the value Host in the <<SaStep>> of the invoking message. This is shown above in Figure 8. This provides the remainder of the CPU-related parameters. The complete process by which a <<SaStep>> would be allocated to a computing resource, which would then be bound to a CPU, is extremely tedious and we have simplified the process to eliminate the intermediate layers to make these capabilities usable in a straightforward manner. We feel that the complete process by which computing steps are bound to logical processors, which are bound to physical CPUs, would not be acceptable to performance analysts because of its complexity. We recommend the simplified approach above to promote the use of this approach when performing a schedulability analysis on well-defined collections of CPUs. If the process is overly complex, neither MARTE nor schedulability analysis will be widely used.

This simplification makes sense because the duration and the execution time in the <<SaStep>> stereotype of the UML model are already calibrated for a specific CPU. The nonlinear effects of changing the memory size, cache refresh policy, bus speed, or any other property that would affect the execution time would have an extremely indirect effect on that execution time. The only meaningful way that people use RMA and scheduling tools is to understand the CPU and its execution nature for an application either by profiling observation or by having highly analogous apriori reasoning and experience data. For scheduling purposes and RMA, it makes no sense to go through the tedious allocation process and CPU subcomponents

specification and characterization as shown in the MARTE examples in the OMG specification. The data for meaningful execution timing entered into tools like RapidRMA™ will be obtained by profiling. Since this is the case, the CPU can be abstracted away from the timing. The timing is in the profiling and observation of executing a standalone component on the specific processor in question. The only additional properties we need to use to characterize the CPU within the RAM model are SpeedFactor, SchedPolicy, SchedPriRange, and CnTxtSwT.

The instance of *periodic* on the sequence diagram is invoked in a slightly different manner. *Message_1* is a self directed message on the timeline for the periodic task. It does not come from the environment. It is still implemented in the same manner as *message_0* was in the example. It is realized as an event within UML and we have applied the same stereotypes from MARTE, namely, <<TimerResource>> and <<SaStep>>. We obtain the same annotation capabilities for occurrence pattern, deadline, and execution duration as required. This causes repetitive execution instances of *periodic* in the desired manner. The *buffer* object in the diagram represents a mutually exclusive resource. The stereotype for <<SaSharedResource>> has been applied to the *buffer* class. This provides the additional parameters necessary for resource characterization like acquisition and deacquisition time. The list of tags is shown below in Figure 9:

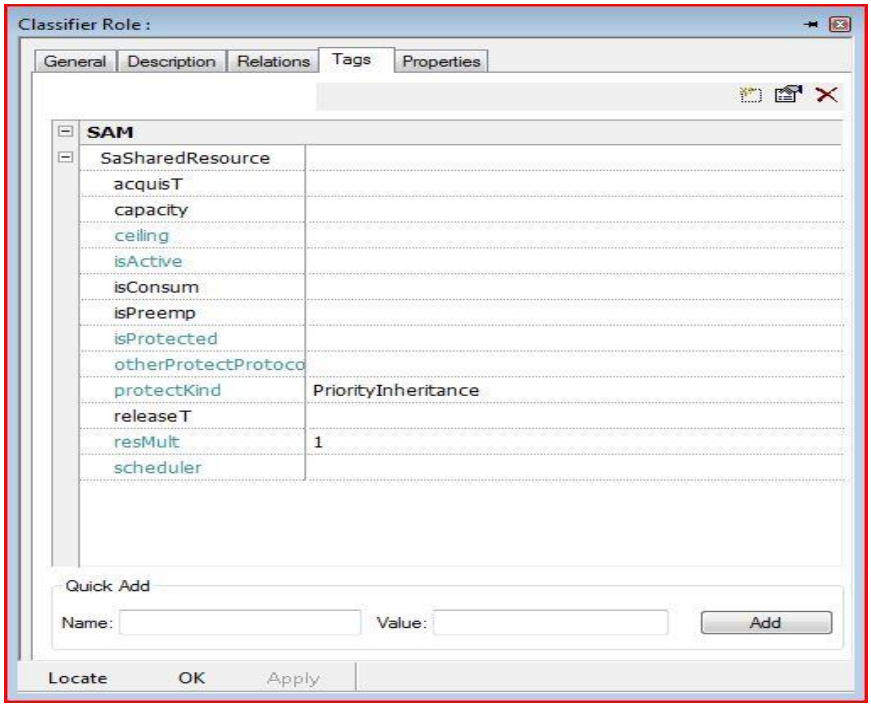

Figure 9 Tags for Mutexed Shared Resources

The shared resource does not have its own thread of execution so there are no invocation timing characterizations required about initiating it. Note that the operations within the *buffer* object named *get* and *put* are drawn this time with closed arrows rather than open arrowheads. The significance of the closed arrow shows that this operation executes in the control thread of the caller and operates as a passive procedure call.

Note that the stereotype <<SaStep>> has been applied to the *get* and *put* operations in the *buffer* class rather than on any connectors in the sequence diagram. This characterizes each usage of the resource in a consistent manner. See Figure 10:

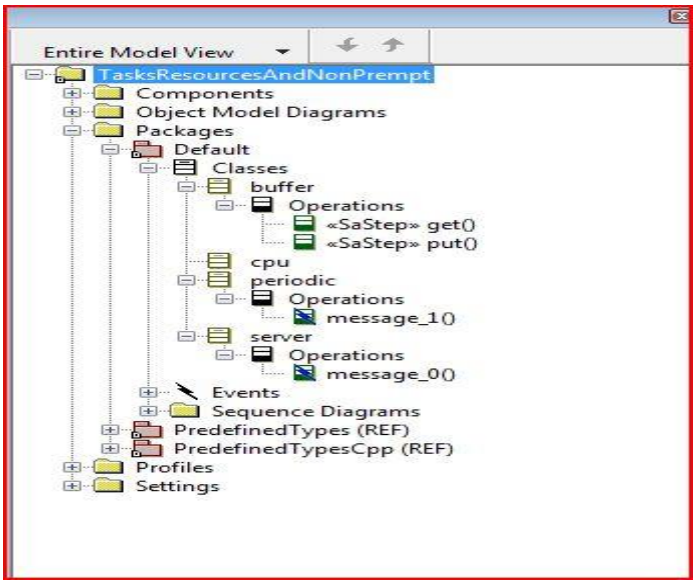

Figure 10 <<SaStep>> applied to mutexed class operations

The time spent in the resource operation is added as execution work to the caller's thread for timing and schedulability purposes. This is exactly the desired effect. The total work is now the duration of work done as specified in the <<SaStep>> of the caller when added to the duration specified in the <<SaStep>> of the operation within the buffer resource being called by the caller. For brevity, the <<SaStep>> tags display has not been repeated here in the document. It would be identical to the one shown previously for *message_0*.

Second example -- Task dependency on multiple CPUs

This example illustrates computation dependency. Starting one task depends upon completion of another. *Task2* cannot begin execution until *Task1* is complete. This example executes on two CPUs. The sequence chart is shown as Figure 11 below.

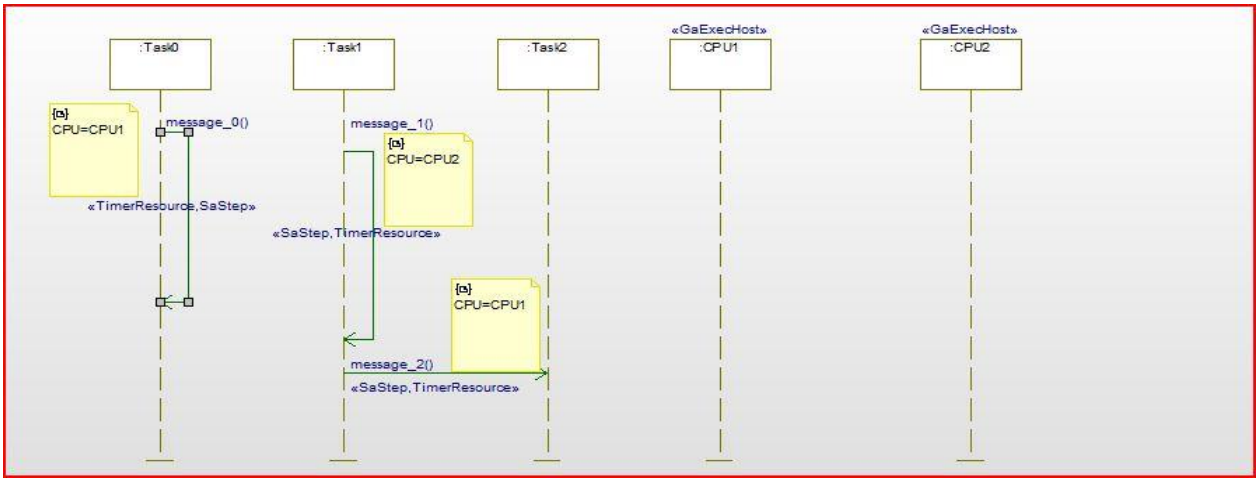

Figure 11 Multiple CPUs Example

Three tasks and two CPUs are shown in the diagram. Two tasks (*Task0* and *Task1*) are invoked periodically as in the previous example. This is done by self-directed messages for each task (*message_0* and *message_1*) by applying the <<SaStep>> and <<Timer Resource>> stereotypes. Application of the stereotypes gives a place for the occurrence pattern, the deadline, and the duration of work that occurs for each invocation to be

specified as tags for the message. In addition, a local tag designating CPU as described previously in the document has been added to each message. *Message_0* has a CPU property equal to *CPU1*. *Message_1* has a CPU property equal to *CPU2*. This associates the execution of each instance with residency on the correct CPU.

The CPUs are drawn on the sequence diagram to represent the deployment platform. They are not explicitly called on the diagram. They are accessed by a reference from the value of the CPU property of each message within the local CPU tag.

Message_2 originates after the completion of the *Task1* instance invoked by *message_1*. This correctly indicates that the beginning for an execution instance of *Task2* depends on the completion of an execution instance of *Task1*. The timing properties for *message_2* are obtained from the timing properties of the calling thread and *Task2* occurs with the same periodicity as the caller. The amount of work for each instance of Task2 is entered as the duration in the <<SaStep>> property since that stereotype has been applied to *message_2*.

Implementation of tools and a user interface to support this type of invocation dependency needs to ensure semantic consistency between calling invocations and other subsequent downstream dependent task invocations. In the sequence diagram, the stereotypes for <<SaStep>> and <<TimerResource>> have been applied to *message_2*. The parameters relevant to the deadline and occurrence pattern for *message_2* causing invocation of a *Task2* instance are obtained from the execution properties of *Task1*. This ensures that the task dependency chain will have consistent periodicity. This is semantically required in a dependency relationship. *Message_2* is the only message in the example which creates a task dependency. This is because it does not originate directly from an event or from the environment. It originates as a consequence of the completion of Task1 since it is the next thing sequentially in the execution chain of *Task1* execution as defined in the sequence diagram.

Analysis of this sequence diagram will ensure that the proper dependency and ready time computations are obtained from the invocation path specified within the sequence diagram. This is all that needs to be done to characterize multiple CPUs, and task invocation dependency across or within CPUs as desired.

Third Example -- Multi CPU Termination Dependency

It is often necessary to show dependency between tasks defining their organized termination. Previous dependency examples and current support within schedulability tools shows dependency for invocation termination. This example shows how to use mutually exclusive resources to obtain controllable semantics for synchronized termination of multiple tasks.

The behavior that we desire to achieve is most easily described in an activity diagram. This is because one of the primitive control forms in an activity diagram is a synchronization bar to gather together parallel threads into a downstream sequential thread. The activity diagram to convey the synchronization that we desire appears below as Figure 12:

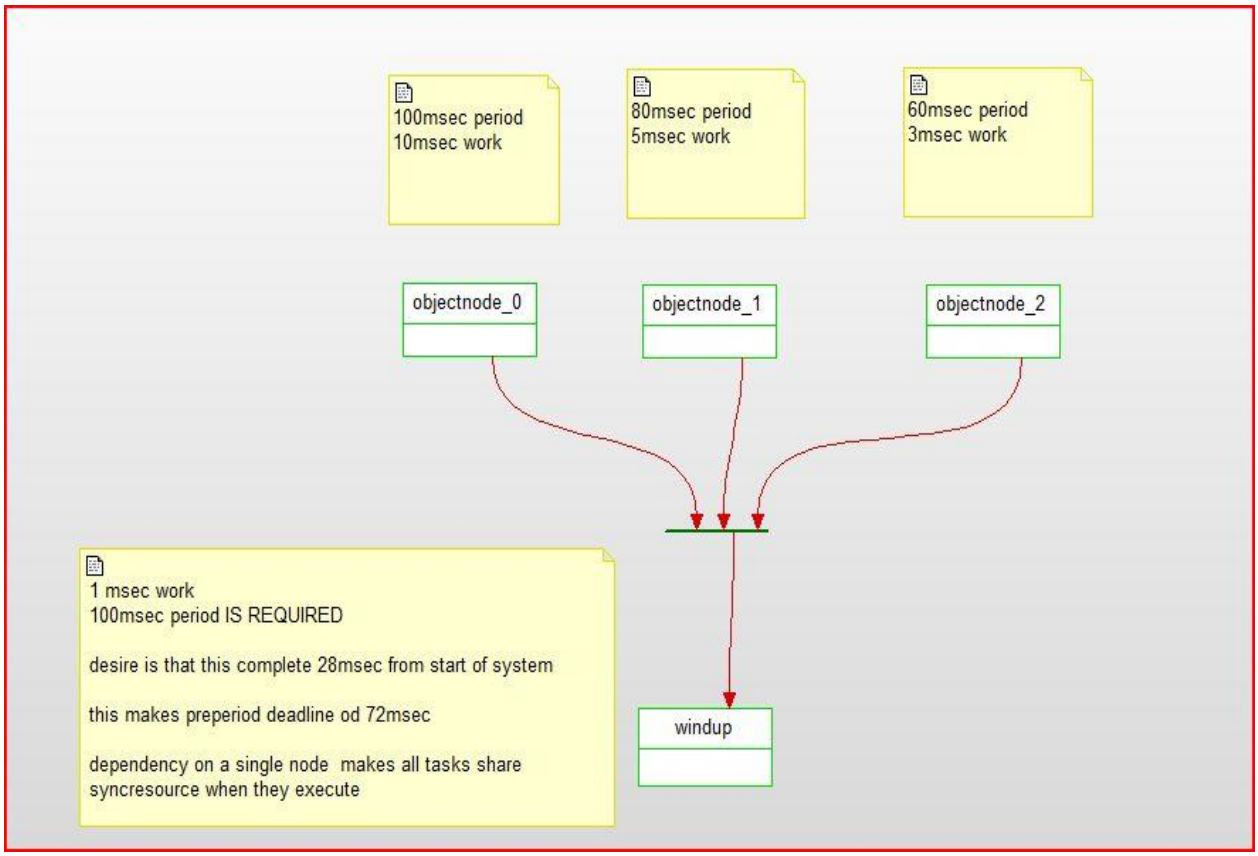

Figure 12 Activity Diagram describing synchronized tasks termination

In a sequence diagram, three tasks (*Task0*, *Task1* and *Task2*) are all started in a normal manner as before using occurrence pattern, deadline and duration parameters to characterize each executable instance. Once again we will use <<SaStep>> and <<TimerResource>> to generate places to hold those values within the originating message.

RMA does mathematical determination of worst-case response time. It does not perform calculations to decide "what goes before what" but only guarantees that the sequence of things that must be non-overlapping can complete within the obtained analytical worst-case response time. This provides us with a means to generate the dependency semantics needed within the activity diagram. All three originating tasks must be completed before the consequential *windup* task has begun. For this example we will use one CPU for simplicity. This example may be extended to multiple CPUs in a straightforward manner. Since there is one CPU, the effect of the dependency relationship shown in the activity diagram on one CPU means that all four tasks must be able to execute in a sequential order. This is because the *windup* task cannot begin until *Task0*, *Task1* and *Task2* have completed. In addition, these three tasks all compete for the CPU and must all vie for the CPU resource between themselves. We will define a mutually exclusive resource called *SyncResource*. This resource will be exclusively held by each task during the entirety of its execution. By defining and utilizing resources such as this throughout the execution of the task, we can define run-to-completion semantics because any other task which might preempt will also need the resource. As a consequence those potentially preemptive tasks are unable to run until the resource is released by the completion of execution for the task currently holding and locking that resource. The sequence diagram for this example is shown in Figure 13 below.

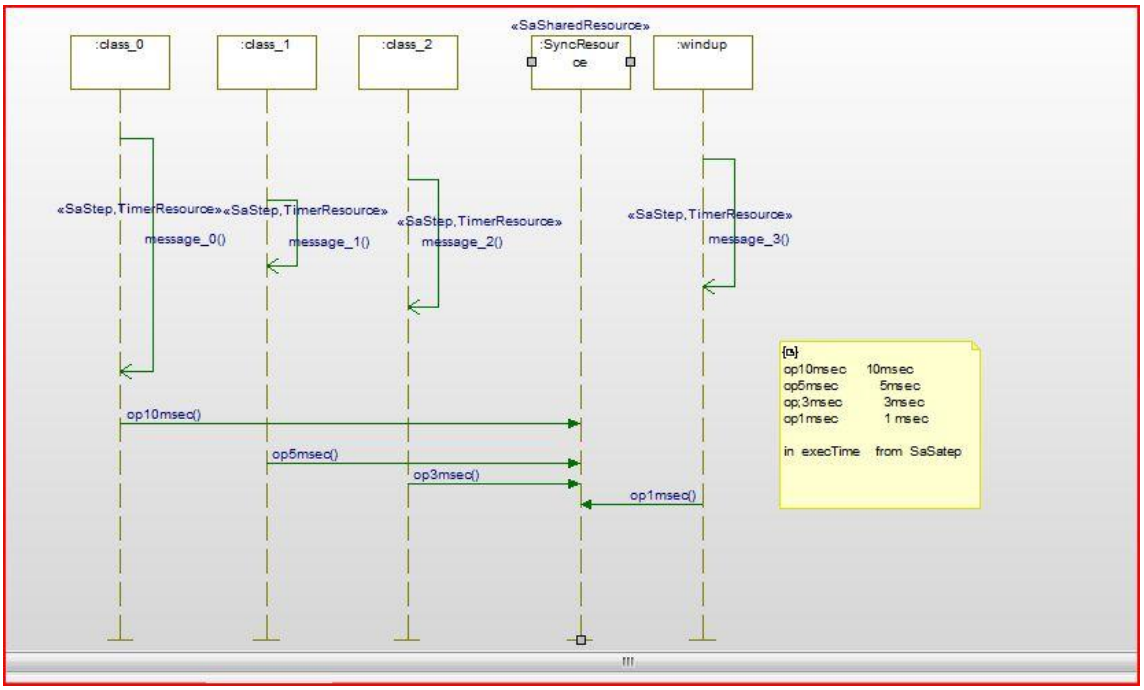

Figure 13 Sequence Diagram describing synchronized task termination

The sequence diagram shows four tasks that are independently invoked by self-directed messages. The usual stereotypes (<<SaStep>> and <<TimerResource>>) have been applied to the messages to specify occurrence interval, deadline, and duration. The <<SaSharedResource>> stereotype has been applied to the *SyncResource*. Since the *windup* task is dependent upon the termination of the other three ancestor tasks, its repetition period cannot be any smaller than the largest period of any of the ancestor tasks.

The *SyncResource* is a mutually exclusive resource. One caller at a time locks that resource and in turn the arbitration policy set within the scheduling context (for example priority ceiling, highest lockers, or priority inheritance) will determine the priority by which the calling thread executes while holding the lock resource. This behaviour is indicated by the use of a closed arrow rather than open arrow. This designates a passive procedure call rather than multiple execution schedulable threads.

Four operations have been defined on this resource to denote their use by each client for an appropriate amount of time. Each operation has had the stereotype <<SaStep>> applied to the operation. As a consequence, there is a tag for ExecTime available with each operation. This serves as the place to enter the duration of time associated with each call to the operation. That is the time for which the resource will be exclusively held by the caller.

3. CASE STUDY

In order to experiment the previously described approach, we have defined a case study coming from a Software Dynamic Design description of a Flight Warning system. We have extracted the main features of the existing dynamic architecture in order to build a simplified but representative subset and to prove the suitability of our approach to model it and run timing analysis. Simplifying the model is easier to reduce the high number of shared resources that are used to communicate between these processes like buffers, blackboards, and events. We have also restrained the modelling to a mode called NORMAL that is reached after initialization upon reception of a message from the environment.

This application is based on ARINC-653 standard. It is part of a single partition hosted by a single CPU and is composed of two concurrent processes. First one is periodic and high- priority (called *PP* in what follows),

whereas the other is cyclic and low-priority (called *AL* in what follows). One main action of the task is to send graphical pages to be displayed at the end of each frame. During each period, process *PP* reads I/O, performs some work and writes some data to a buffer, this data is then read by *AL* before performing some computational work. Such system would be schedulable if the cyclic task does finish its task after a given and fixed number of periods. In what follows, we restrain the verification for the case of only one period, which is sufficient to prove the schedulability for any number of periods due to the fact process *AL* is cyclic. However, we will discuss the general case in next section.

First step is to create a sequence diagram to represent corresponding behaviour. For this purpose, we choose to represent the creation of the processes with *normal_synchro* and *normal_synchro_bis* messages, a periodic (same period as *PP*) and a deadline also equal to *PP* period. The sequence modelling follows the described methodology from the previous sections, as shown in following figure 14:

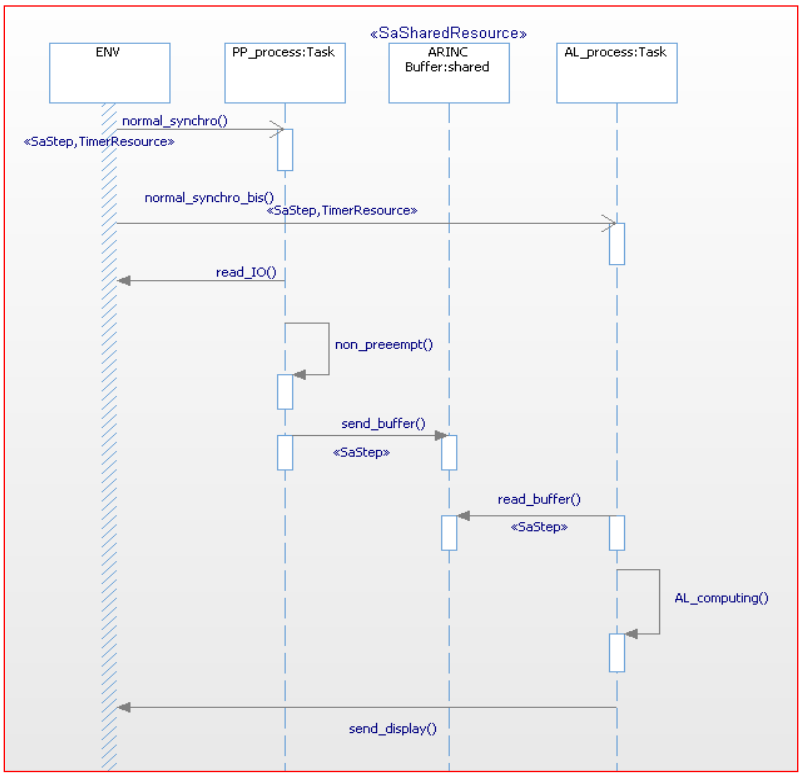

Figure 14 Sequence diagram for a subset of the case study

Next step is to edit the timing properties that are given in the Flight Warning documentation. This concerns mainly periodicity, deadlines and worst case execution time. In the current version of the tool, this is done by setting parameters in the RapidRMA™ "Edit Timing Properties" view, the *normal_synchro* and *normal_synchro_bis* periods are set thanks to parameter SAOccurencePattern and their deadline with SAAbsDeadline. All the WCET for the *non_preempt* and *AL_computing* steps are assigned with RTduration. Note that in this particular single-CPU case, one default processor is automatically generated by the tool. At this point, we are able to run the timing analysis, by passing the required parameters regarding access control protocol and scheduling policy, which is either Rate Monotonic or Deadline Monotonic. With this required information, results show a schedulable system. By increasing the work load to a certain point, the analysis reports an error stating the utilization for the default processor by giving the percentage.

4. FUTURE WORK AND CONCLUSIONS

This paper presents a methodology based on the combined use of MARTE profile and RapidRMA™ tool, through its IBM Rhapsody™ plug-in version, in order to characterize the timing and resources properties necessary to specify a complete set of inputs for a rate monotonic schedulability analysis. Our approach consists in applying a reduced set of MARTE stereotypes. The complex features from MARTE dealing with logical CPU, physical CPU, computing platform, scheduler, allocation, etc. while being totally general and extensible in nature, add difficulty and burden to understanding and use. The choice of the Host tag (which does not require the use of the allocation process) to assign thread instances to specific processors provides an extremely straightforward and easy to grasp capability for a multi-CPU schedulability analysis. Execution timing data is obtained by profiling or historical analysis. The only additional timing characteristics necessary for a CPU to support analysis purposes are normalized clock speed (so that margins can be checked by hypothesizing increases or decreases in CPU speed), and context switch information. A summary table of the stereotypes and their application is shown below in Figure 14:

UML Element	Stereotypes to be applied	Purpose/Reason used
Message	<<SaStep>>, <<TimerResource>>	Occurrence Period , deadline, Execution Time
Resource Operations	<<SaStep>>	Execution Time
Shared Resource	<<SaSharedResource>>	Resource locks and related properties for analysis
CPU	<<GAExecHost>>	Schedule properties, context switch

Figure 14 Stereotypes used and their justification

The feasibility of the described methodology has been demonstrated on a representative case coming from an avionic system. This activity has been driven based on AIRBUS interest for modelling systems in order to be able to perform formal verification at early stages of the design process. This paper shows successful results of the approach when applied to some industrial use case subset. Of course, further work is required before getting to the integration of proposed approach in future software architecture design process. First, there is a need to pursue the in-depth analysis of the MARTE modelling profile. As a matter of fact, this is required to assess how well this approach could fit the industrial design methodology. Another currently ongoing step is to investigate more case studies in order to get a better knowledge of RapidRMA™ capabilities based on different systems from the avionics world. For example, as mentioned earlier, the case study of this paper should actually show a cyclic task spreading over a few periods of *PP* process. In order to deal with this problem, one can assign a sporadic server to the cyclic task. This can already currently be done with standalone RapidRMA™ tool, but remains out of the scope of the described UML based approach of this paper. Possible solutions, from simple but not standard UML to fully MARTE compliant and integrated, have to be assessed.

5. REFERENCES

[1] UML Profile for MARTE: Modelling and Analysis of Real-Time Embedded Systems, OMG