From User Requirements to Tasks Descriptions in Real-Time Systems

Leo Ordinez, David Donari, Rodrigo Santos and Javier Orozco

Instituto de Investigaciones en Ingeniería Eléctrica
Universidad Nacional del Sur - CONICET
Av. Alem 1253 - Bahía Blanca - Argentina
{lordinez, ddonari, ierms, jorozco}@uns.edu.ar

***Abstract*—Real-time scheduling theory has made a great progress in the last decades. From small devices to enormous satellites or industrial plants take advantage of this ongoing research. However, to the authors belief, there is still a gap to fully exploit the benefits of the theory.**

The cornerstone of real-time scheduling theory is the concept of task. Nevertheless, very little is mentioned about how to discover such tasks. Thus, the objective of this paper is to propose a systematic way of describing real-time tasks by means of requirements elicitation. The process begins with the gathering of user requirements through Use Case Diagrams. Then, those use cases are refined to turn them into tasks descriptions. Lastly, tasks descriptions are further refined and brought to a detailed characterization of individual execution flows. This final tasks characterization is made with Activity Diagrams.

Finally, a Line-Follower System is used to exemplify the proposed approach.

***Index Terms*—real time systems; software requirements and specifications; software tools; software testing**

I. Motivation

Real-time software systems are systems that have strict timing constraints in their specification. This is, not only they have to be functionally correct, but they also have to meet certain intrinsic timing restrictions, that arise from the problem definition itself. Therefore, the analysis of this kind of systems implies a careful study of functional and non-functional requirements in a stage as early as possible.

Within real-time systems theory, there is a particular branch that deals specially with the meeting of timing constraints: *the real-time scheduling theory* [1]. The cornerstone of real-time scheduling theory is the concept of task. Basically, a real-time task is a software component which has a particular purpose within the system. In Jackson's terms [2], the previous statement can be rewritten as: *a task is a specification of a software requirement extracted from the application domain.* In this manner, a real-time system is said to be composed of tasks, that perform the necessary functions to fulfill the user requirements. Thus, simplistically the complete set of tasks answers the question: *What does the system do?* Nevertheless, very little is mentioned in the specialized literature about how to discover such tasks. In this sense, the analysis of real-time systems is in most cases a job, where experience plays a determining role. However, experience is not enough to get an optimal analysis, that allows to efficiently obtain a correct design [3].

The implementation of a real-time system begins, like any other software system, with the description of a problem to be solved. This description is not always (almost never) formal and fully detailed. Usually, the description is ambiguous, with little details, not very clear and in colloquial language. This leads to the problem of investigating, inquiring and analyzing many things to finally describe precisely the problem to be solved. However, this is just the problem description. From the problem description to the identification and subsequent behavioral description of the different tasks to be implemented there is still a long way. Moreover, without that identification it is extremely difficult to apply the real-time scheduling theory.

With all, the objective of this paper is to propose a systematic way of describing real-time tasks by means of requirements elicitation. The process begins with the gathering of user requirements through Use Case Diagrams. Then, those use cases are refined to turn them into tasks descriptions. Lastly, tasks descriptions are further refined and brought to a detailed characterization of individual execution flows. This final task characterization is made with Activity Diagrams.

A. Specific Contributions

The main contributions of this paper include:

- ▲ The adaptation of use cases to deal with real-time systems needs.
- ▲ The presentation and discussion of a use case architecture for real-time systems.
- ▲ A method for eliciting tasks through user requirements and use cases.
- ▲ The adaptation of activity diagrams to deal with real-time tasks.
- ▲ A method for describing the behavior of tasks.

B. Paper Organization

After this introduction, the rest of the paper is organized as follows: in Section II, use cases are adapted to handle real-time systems requirements. In addition, a use case generic architecture is developed for this application domain and a method to elicit tasks adjusted to that architecture is also presented. In Section III, the previously mentioned tasks are

refined and their behavior is described in detail. An example of a Line-Follower robot is presented in Section IV to illustrate the proposed approach. Finally, conclusions are derived in Section VI.

II. Tasks Elicitation Through Requirements

In this section, a method for gathering user requirements through use cases is developed. The method shares basic aspects commonly found in the literature [4], [5], [6], [7], [8], but introduces some improvements for dealing specifically with real-time systems. In this sense, use cases are adapted (restricted) to cope with real-time systems needs and a generic use cases architecture is developed for this kind of problem frame. Finally, real-time tasks are elicited from those gathered requirements and briefly described.

A. Restricted Use Cases

Use cases are simply an alternative mean of specifying requirements [9]. A *functional* requirement defines a function of a software system; while a *non-functional* requirement specifies criteria that can be used to judge the operation of a system, rather than specific behaviors. In this sense, a Use Case diagram (UC) [10] describes **what** the system does.

As was mentioned in Section I, applications with timing constraints need a different treatment from other kind of applications. In particular, uses cases as presented in [10] may be too broad to capture requirements for them. Following, use cases are adapted to cope with real-time systems needs.

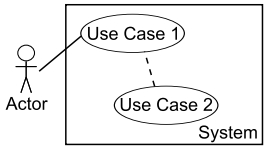


Fig. 1. UC diagram modeling constructs

1) Drawing Use Cases: The easiest and fastest way of using use cases is by drawing them. As stated in the UML standard [10] a UC diagram has four modeling constructs, (see Figure 1): system, actor, use case and communication lines, that express relations between constructs. These constructs are briefly analyzed for the case of real-time systems as follows:

1. *System* construct: For systems that are composed of hardware and software, as the case of real-time systems, it is more appropriate to consider the *System* to be the software and hardware composite. The system construct restricts the scope of the application and includes all of its functionalities. Within the system construct is the purpose of the real-time system.

2. *Actor* construct: An *Actor* represents a specific **role** played by an entity that resides **outside** the modelled system and interacts **directly** with it [4]. As a rule of thumb, any entity that interacts with the system but over which there is no control is a possible actor.

3. *Use case* construct: The functionality of a system is defined by different *use cases*, each of which represents a specific flow of actions. A use case should focus on the **purposes** of the system. This way, the total collection of use cases will actually form the complete functionality of the system under specification.

Item	Description
NAME	*The use case name*
ACTOR	*The actor involved*
DESCRIPTION	*What does the use case do?*
EXECUTION FLOW	*Normal execution flow*
EXCEPTION FLOW	*Exception execution flow*
MONITORED VARIABLES	*Explained in Section II-C*
CONTROLLED VARIABLES	*Explained in Section II-C*
NFR	*Non-functional requirements (specially timing constraints)*

Fig. 2. Use case template

It is worth noting, that a use case can be initiated internally by the system (*e.g.*, according to time) and not only externally by an actor. And it can describe internal functionality of a system and not only its external behavior.

4.a. *Include* relation: This stereotyped relation is used to point out that a use case has some kind of dependency on another use case to achieve its goal. Two kinds of dependencies are identified in real-time systems analysis: *functional dependency* and *sharing dependency*. In the first case, the execution of a task cannot begin until the execution of another task has finished. While in the second kind of dependency, both tasks can run concurrently, but they share a resource whose access is allowed in a mutually exclusive way.

4.b. *Extend* relation: As explained in [11], extension use cases allow to add new behaviors into existing use cases (which are called the base, or extended, use case). The use-case technique is formulated such that the new behavior is described totally separate from the base use case. However, this may not be entirely true in real-time systems, since generally the extended use case covers the base one. Thus, the base use case is only taken into account, in the modeling process, for completeness and clarity reasons. In this sense, the extend relation might be the more confusing one and its use is encouraged only when the extension comes out naturally or due to a refinement process of use cases.

4.c. *Generalization* relation: This kind of relation is only allowed between actors and its semantics are similar to those of the well-known inheritance relationship of object-oriented programming.

2) Writing Use Cases: Use case diagrams presented above lack of many details relevant to the analysis of the system under construction. Although being extremely useful in a first stage, diagrams need to be refine and expanded to fully get the most out of use cases. The way of achieving a higher level of detail is by *writing* use cases. This way of viewing use cases is further explained in [9], [5], [11].

Despite the fact of being some notations more accepted than others, there is no standard when writing use cases. In this paper, a template is proposed to cope with the requirements of a real-time system. The template is shown is Figure 2.

B. Use Cases Architecture for Real-Time Systems

1) Real-Time Systems Generic Architecture: The functioning of a generic real-time system is depicted in Figure 3. The framework is composed by one processor and a set of hardware and software resources all of which are managed by the real-time kernel. Real-time tasks require the processor and establish a quality of service (QoS) contract to use it. It is worth noting that the QoS contract represents non-functional requirements. The resources may be used by the real-time tasks that request them and the real-time kernel provides the mechanisms for controlling those accesses. Finally, the kernel allocates the processor to the real-time tasks according to its QoS contract.

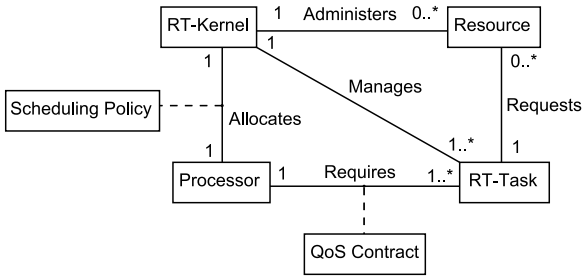


Fig. 3. A Real-Time System Framework

Although the framework is expressed as a set of classes, the model is not restricted to object-oriented technologies for its implementation [12], [13], [14].

2) Capturing Non-Functional Requirements: As explained in Section II-A, use cases capture the functional requirements of a system. However, a system, in general, has certain constraints that are not functional. These kinds of constraints are Non-Functional Requirements (NFR). While functional requirements describe *what* the system *does*, the set of NFR describe *how* the system is supposed to *be*. With this in mind, it follows that NFR must be taken into account when analyzing a particular system. In this sense, the UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) [15] is adopted in this paper to capture NFR.

The MARTE profile is based on *stereotypes*, that provide the necessary capabilities to model domain-specific concepts. In addition, stereotypes may have typed properties called *tag definitions*, that represent attributes or relations; and *constraints*, that represent restrictions.

When analyzing a particular system, generally an analysis method does not map one-for-one to that particular system. In this sense, all analysis methods use a simplified/abstract view of the system to analyze, which focuses on those aspects that are relevant to the associated analysis technique [16]. Thus, to the aims of this paper, the MARTE profile is partially adopted, just to express timing constraints typical in real-time systems. The limitation on the usage of the MARTE profile is done because this paper is focused on an early stage of the software development lifecycle and there are aspects of the system that might not be clear yet. Figure 4 shows the stereotypes used in the proposed approach.

3) Use Case Architecture: In this section, the real-time systems framework presented in Section II-B1 is mapped to use case diagrams and a use case architecture for real-time systems is presented.

Stereotype	Tag	Example
Transaction	isSchedulable	≪transaction≫ RTTask1 {isSchedulable = true}
Trigger	pattern period	≪trigger≫ TaskTrig {pattern = 'period', period = 10ms}
Response	deadline	≪response≫ RTTask1 {deadline = 9ms}
SaSharedResource	isPreemp isConsum	≪SaSharedResource≫ ShVariable {isPreemp = False, isConsum = False}

Fig. 4. Subset of the MARTE profile.

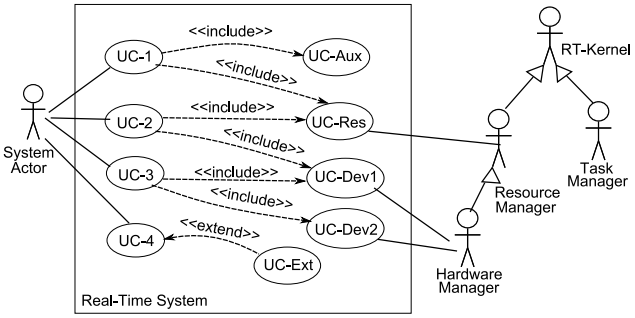


Fig. 5. Use case architecture for real-time systems.

Figure 5 depicts the proposed use case architecture. On the right-hand side of the figure, the actors that represent the real-time kernel structure are shown. As can be seen, the only actors that directly interact with the system are the *Resource Manager*, which is in charged of handling shared software resources, such as shared variables and inter-task communication mechanisms; and the *Hardware Manager*, that provides the necessary services to interact with the hardware platform.

On the left-hand side of Figure 5, an example of system actor is depicted. This actor is in fact the one that requests certain services from the system through the different use cases. Concerning uses cases, they can be divided in two groups. The first group (which is composed of *UC-1*, *UC-Aux*, *UC-2*, *UC-3*, *UC-4* and *UC-Ext*) consists of the use cases requested by the system actor. These use cases are the actual functional requirements of the application being developed. Whereas, the second group (*i.e.*, *UC-Res*, *UC-Dev1* and *UC-Dev2*) is responsible for providing real-time kernel services to the first group of use cases. This last concept is a subtle change in the point of view of use cases, since real-time kernel actors are not benefit from the system through their use cases, but they provide those benefits to the system by means of their use cases. Nevertheless, this is just a different interpretation, the semantics of use cases remain actually unaltered.

With respect to the relationships among use cases, five types of them are distinguished (from the top of Figure 5 to its bottom):

- The first type of relation is the one that expresses the use of an auxiliary use case (*UC-Aux*) by a base use case (*UC-1*). This relation models a common interpretation of the *include* relation, which is the calling to an auxiliary function. That is, *UC-1* needs *UC-Aux* in order to achieve its goal.
- When two or more use cases need to access the same resource (as the case of *UC-1* and *UC-2*), that resource is provided by the *Resource Manager* actor. Thus, both use cases include the use case related to that share resource (*UC-Res*).
- The previous situation is analogous to that of a shared hardware service. This relation is exemplified by *UC-2*, *UC-3* and *UC-Dev1*.
- When a use case (*UC-3*) requests for a hardware service (*UC-Dev2*) that is not requested by any other use case, the relation is straight. This relation is similar to that of the first item, with the difference that in this relation both use cases do not belong to the same group of use cases (see the classification presented before this itemized list).
- Finally, there are analysis situations in which a use case (UC-Ext) extends the capabilities of a base use case (UC-4). This situation is to be handled carefully, since it is very error prone.

C. Eliciting Real-Time Tasks from Use Cases

In a real-time system, generally there is no much interaction between a human and the system. Nevertheless, if the system has significant external visible behavior, then use cases are useful to capture it. On the other hand, since real-time systems are composed of many tasks that do not have an external behavior, use cases can also be useful to describing an internal functionality of the system and not just its external behavior. Following, a series of steps are presented to build use cases and elicit real-time task from them.

1) **Define the system**
   The first thing that must be defined is what kind of system is going to be built. By defining it in the first place, the scope of the system can be narrowed. In order to clearly and unambiguously define the system, the first thing to do is naming it, then define its purpose and finally write a short description of it.
2) **Find the actors**
   Any entity that is outside the control of the system, but interacts with the system is a possible actor. Usually, the external entities that interact with a system are human beings or external devices. The key to distinguish a user from an actor is the role that the last one plays. A role focuses on the responsibilities not in the person. On the other side, not every external device is an actor. A sensor or actuator do not use the system, consequently they are not external systems and then are not actors.
   In order to find actors, every external entity that needs to interact with the system has to be identified. Then, each actor must be named and described precisely.
3) **Find the use cases**
   Each use case describes what the system does. Thus, the key to find use cases is to identify the largest functionalities of the system. Particularly, for real-time systems a list of actor's tasks must be created and visualized as use cases. Tasks are characterized as goals that actors want to achieve. In many cases, the completion of those tasks has an effect in the environment.
   Naming the use case is essential. The name of the use case must reflect the intention of the actor. A simple and effective way of naming use cases is following Zhang's rule [6]: "*The* `<system's name>` *is required* [*by the* `<actor's name>`] *to do* `<use case's name>`." As stated in Section II-B3, there are two groups of use cases: the ones that benefit an actor and the ones that an actor provides to benefit others use cases. In general, the second group is mostly composed of those use cases provided by the real-time kernel actor and its specialized ones. However, there are some exceptions as the case of auxiliary use cases or extended ones. Concerning the first group, use cases that benefit actors, they are the actual tasks that compose the system. Whereas, the others are auxiliary functions. When naming both use cases a distinction must be imposed. Consequently, the following rule is applied: for functions <*object*><*verb*> and for tasks <*verb*><*object*>. Thus, the use case *Sensor read* is a function and *Adjust position* is a task.
4) **Define inputs (*monitored variables*) and outputs (*controlled variables*) for each use case**
   In general, monitored variables are associated to sensors or external information and controlled variables are associated to actuators or internal information. However, some variables can be monitored and controlled at the same time.
5) **Solve overlaped use cases**
   Two or more actors can require the same use case from the system. Mostly, the difference is that each actor has a distinct interface or communication path. In order to solve this situation, two alternatives are given:
   - Leave one single use case, that is shared by many actors. In this case, a mechanism to control the access is needed, since there might be several paths from several actors to reach the use case.
   - Split the use case in as many ones as actors trying to access the original one.

   Note that this step is referred only to use cases belonging to the group that benefits actors and not to those that offer benefits.
6) **Write use cases in textual form**
   A use case is a sequence of actions. This flow of actions is complete and meaningful. Base on the template of Section II-A to write a detailed description of each use case. In this step try to be more specific than in any previous one.
7) **Refine tasks according to variables**
   Taking into account the monitored and controlled variables of Step 4, refine each use case text corresponding

to a task.

8) **Define non-functional requirements**
   In this case, focus non-functional requirements only on timing constraints and shared resources. Thus, define for each task: deadline and period; and for each shared use case in the diagram, characterize it. The definition of non-functional requirements has to be done with the MARTE profile.
9) **Balance tasks**
   In order to balance the complexity of tasks, every use case must be revisited and carefully analyzed. By balancing tasks the system will not have some tasks extremely complex and some almost trivial. But, they will have an even level of complexity. This is, of course, unless the system imposes to do the opposite.
   A way to discover that a task is not actually one but two different ones is through timing constraints. If a use case has, for instance, two deadlines is natural to think that the supposed task is not one but two. Thus, it is necessary to refine it and rewrite it.

It is worth mentioning, that the proposed method is iterative and must be repeated until reaching a satisfactory level of detail.

III. Task's Behavior Description

Once requirements are captured and tasks are identified, the next step is to describe those tasks precisely and unbiguously. This is, requirements must be specified by means of tasks descriptions. In order to describe the behavior of a task, firstly Activity Diagrams need to be adapted and restricted to satisfy the needs of a real-time system. Once that new characterization is done, use cases can be easily translated to Activity Diagrams in order to precisely describe their behavior.

A. Restricted Activity Diagrams

An Activity Diagram is composed of several modeling constructs and it describes a logical unit of work [10]. It can be broken down in *actions*. An action is the smallest unit of work that is not decomposed any further. The sequencing of actions is controlled by control flow edges. Actions are joined by edges that represent process flows or events.

Before presenting the restrictions imposed to Activity Diagrams, the why for those *restrictions* must be clarified and explained. In the first place, there are certain functionalities (constructs) that to the aims of this paper are useless. For instance, object-oriented constructors. In the same line of reasoning, there are situations not allowed in the approach. For instance, those that imply sending or receiving signals or events.

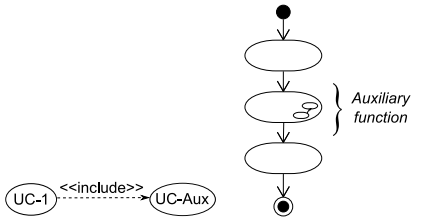
In general, real-time tasks are a finite, sequential control flow that has a particular purpose. Having this in mind, in the approach proposed in this paper neither sending nor receiving signals or events is allowed; since synchronization is achieved by means of shared variables. In addition, anything related to objects management is also not permitted, since the approach is not restricted to object-oriented programming. Concerning *fork* and *join* nodes, they are subjected to a special treatment. This nodes are used to show the interaction between the real-time tasks and the real-time kernel. This interaction can be done basically because of a hardware system call or because two or more tasks share a resource. Finally, in the case of auxiliary procedures, the use of *sub-activity* states is encouraged.

The main advantages of having restricted activity diagrams are: simplicity in the model and ease in the automatic code generation from the graphic model.

B. Describing Tasks

The process of describing tasks is divided in two parts. In the first part, the mapping between situations that arise in Use Case Diagrams and their corresponding ones of Activity Diagrams is made. This is done by showing how the Use Case Architecture for Real-Time Systems is translated to Activity Diagrams. In the second part, the detailed process of mapping each use case to an Activity Diagram and the description of its behavior is done.

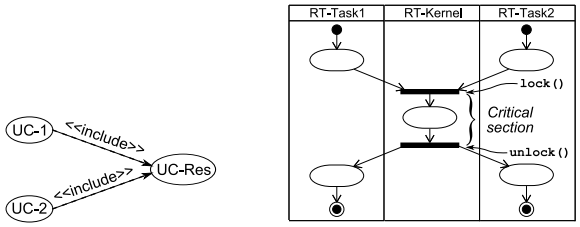
Figures 6 to 10 show the previously mentioned mapping. In particular, the way of expressing the use of an auxiliary function is depicted in Figure 6. In order to use a shared resource or a device by two tasks, there must be some kind of synchronization mechanism imposed by the real-time kernel. Moreover, the real-time kernel is the one that actually carries on the update on a shared resource or the hardware service by means of its Application Programming Interface (API). These two situations are depicted in Figure 7 and in Figure 8, respectively. On the other hand, when only one task asks for a kernel service, there is no need for synchronization. Thus, the mapping is similar to that of an auxiliary function, but making clear that the requested service belongs to the real-time kernel (see Figure 9). Finally, the last mapping has to do with the extension relation. In Figure 10 can be seen that the use case that is in fact translated to an activity diagram is the extended and not the base one. This follows from a previous discussion about the *extend* relation.



(a) Use case. (b) Activity diagram.

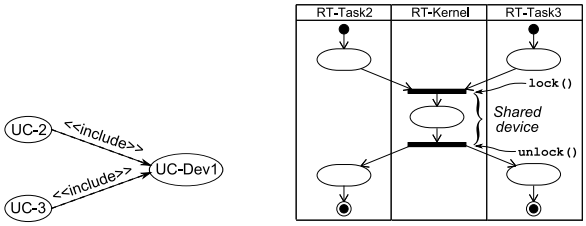
Fig. 6. Auxiliary function mapping.

Before starting with the process of describing behavior itself, it would be interesting to classify the elicited tasks that arise from the use case analysis phase. This classification might be very useful in the following stages of the development process and can also be helpful to detect the occurrence of mistaken analysis choices. In this respect, Zhang [6] identifies three kinds of objects that come from use cases:



(a) Use case. (b) Activity diagram.

Fig. 7. Shared resource mapping.

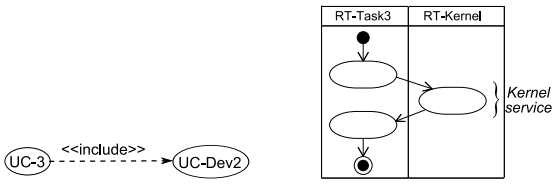


(a) Use case. (b) Activity Diagram.

Fig. 8. Shared device mapping.

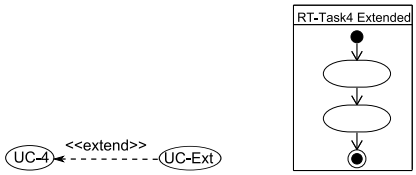
1) control objects, to coordinate the interactions between the system and the actors; 2) entity objects, that execute actions; and 3) interface objects, that communicate with external actors. In this paper, there is no restriction to object-oriented programming. However, Zhang's classification can still be applied with just some subtle adaptations. Thus, the classification is as follows: control objects are related to mutual exclusion among tasks and task management (creation, destruction, scheduling, activation, etc.). Those control objects are, in this paper, real-time kernel primitives that deal with software issues. These primitives differ from those of the interface objects in that the last ones are related to hardware issues. Finally, Zhang's entity objects are actually tasks themselves.

The process of describing tasks is based on the one proposed



(a) Use case. (b) Activity diagram.

Fig. 9. Device usage mapping.



(a) Use case. (b) Activity diagram.

Fig. 10. Extension mapping.

by Kimour and Meslati [7], but having into account the restrictions and characteristics of the model proposed in this paper. The method is iterative. During the process, the use cases texts may be looked through, changed and enhanced to exhibit a greater degree of accuracy and details. In addition, detected ambiguities, inconsistencies and errors are removed.

Kimour and Meslati proposed a three step method, in which each step consists of an iteration. Following, the description and necessary adaptations to that method are presented:

Activity Iteration: The use case text is transformed into an activity diagram. It is important to model the normal flow of actions first and then integrate the exception flow. Also, in this iteration, possible concurrency between actions (*i.e.* the cases of Figure 7, 8 and 9) are specified. Finally, guard conditions on edges or within decision nodes are placed.

Action Iteration: Each action is specified according to the criteria used to distinguish auxiliary functions from tasks and enhancing the level of detail by taking into account the API provided by real-time kernel. Besides, for each action it is necessary to specify the input and output variables that are involved. Guard conditions should be revisited and refined. It is worth noting, that an action should refer to no more than one system unit or one actor.

Consistency Iteration: The activity diagram is checked for internal completeness and consistency. Moreover, timing constraints are further checked and refined according to the MARTE profile.

As was previously mentioned, the above is an iterative method and as such each special iteration (*i.e.* activity, action and consistency) must be further repeated until achieving a satisfactory level of detail.

IV. Example: Line-Follower Robot

In order to exemplify the proposed approach, a simple application is presented: a Line-Follower Robot. The purpose of this system is to control a robot, so that it follows a black line painted in a white floor. The robot has two light sensors and two motors that are controlled by the application. In addition, the robot has another sensor (a touch sensor) that is used to check for obstacles and avoid them. The system was actually implemented in a Lego Mindstorms platform. It is worth noting, that the application to control the robot was running on top of an Real-Time Operating System developed by the authors [17]. Thus, the generic architecture proposed in Figure 5 fits the needs of this particular application.

In Figure 11 the use case diagram of the Line-Follower System along with its MARTE profile annotations is depicted. As stated above, the analysis begins by capturing user requirements through use cases. From that analysis follows that there are three main tasks in the system: 1) *Adjust Position*, 2) *Check Collision* and 3) *Move Forward*. This last task, *Move Forward*, is actually an initialization task. This is, the robot is supposed to start with the black line below it and the light sensors pointing at the white floor. So, the robot moves forward for a certain amount of time following the black line. *Move Forward* is in fact a one-shot task. Task number 2), *Check Collision*, has

an auxiliary procedure (*Obstacle Dodge*[1]), which is called in case that the calling to *Sensor Crash Read* returns that there is an obstacle. It is worth noting, that both use cases *Move Forward* and *Obstacle Dodge* include the use cases related with the motors, however in order not to bother with so many arrows they are omitted in the figure. The same criterion is used to annotate the diagram with the MARTE's stereotypes (*i.e.*, just a couple of use cases are annotated).

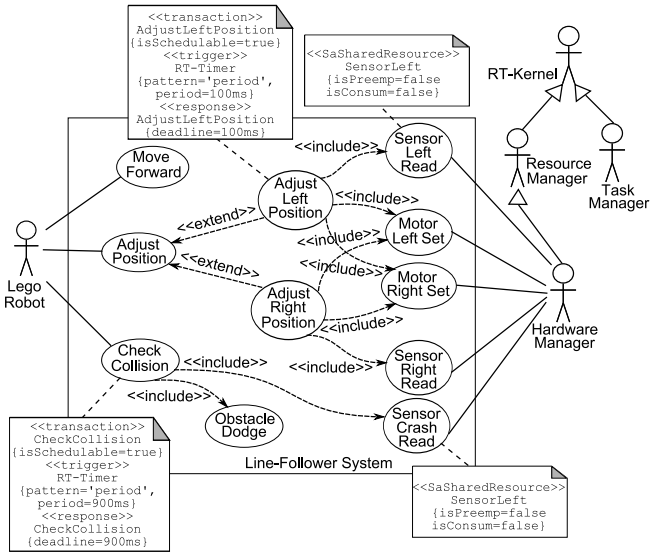


Fig. 11. Use case diagram of a Line-Follower System

During the process of tasks elicitation, it came up that task *Adjust Position* had to be refined and balanced since its input and output variables along with its time constraints demanded it so. Thus, *Adjust Position* was actually refined into two different tasks: *Adjust Left Position* and *Adjust Right Position*. They both do not share any of the two light sensor, so they respectively call for the RT-Kernel services *Sensor Left Read* and *Sensor Right Read*. However, both task share the two motors. This is because in order to, for example, turn left the application must stop the left motor and advance the right one. It is worth noting, that the mutual exclusion mechanism for the two motors must be provided by the RT-Kernel since it manages the hardware services.

The next step in the process is writing use cases. In Figure 12 and Figure 13 two representative cases are shown: *CheckCollision* and *AdjustLeftPosition.*

Once use case texts are written, they are translated to Activity Diagrams. Note the use of the slanted font style and the removal of blank spaces in the task's names. This is done to distinguish an Activity Diagram name from use case name (which uses an italic font style). The task *CheckCollision* is shown in Figure 14(a). Note that its execution flow is very simple, but it has two noticeable characteristics. In the first place, *CheckCollision* uses a touch sensor to detect whether the robot crashed or not. Thus, it needs to call the Real-Time

[1]Note the usage of the rule for naming tasks and auxiliary functions presented in Section II-C.

Name:	Check Collision
Actor:	Lego Robot
Description:	Detects whether the robot crashed into an obstacle and if so avoids it.
Execution flow:	1.a. Determine whether an obstacle was crashed. 2. Exit use case.
Exception flow:	1.b. The robot is crashing into an obstacle. 1.b.1 Avoid the obstacle. 1.b.2 Exit use case
Monitored variables:	`crashSensor`
Controlled variables:	None
NFR:	• The checking for collision should be done periodically with a frequency lower than a second. • The obstacle avoidance manuever should be able to avoid an object of at most $32cm^2$.

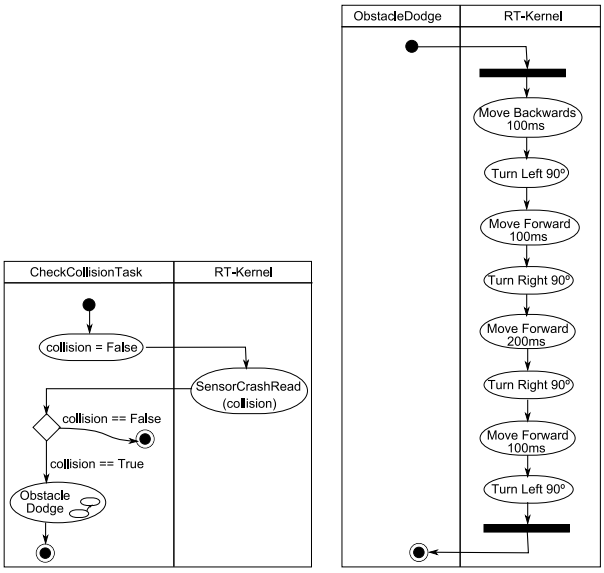
Fig. 12. Check Collision Use Case.

Name:	Adjust Left Position
Actor:	Lego Robot
Description:	Adjusts the Lego robot left side to be over the painted line.
Execution flow:	1. Determine whether the robot is over the line. 2.a. The robot is correctly positioned. 3. Move forward. 4. Exit use case.
Exception flow:	2.b. The robot is not over the line. 2.b.1. Turn robot to the left. 2.b.2. Move forward. 2.b.3. Exit use case.
Monitored variables:	`leftSensor`
Controlled variables:	`leftMotor` `rightMotor`
NFR:	• Left position should be adjusted periodically every 100ms. • The turning left manuever should not exceed 10 degrees. • Moving forward should not exceed a distance of 1cm.

Fig. 13. Adjust Left Position Use Case.

Kernel service *SensorCrashRead*. That interaction between the user task and the Real-Time Kernel is shown by, the use of partitions that separate the two different spaces. However, in this case no synchronization mechanism is used since the touch sensor is not shared by any other task. The second characteristic has to do with the calling to an auxiliary function. The calling is shown by the use a sub-activity state and the actual auxiliary function is depicted in a separate Activity Diagram (see Figure 14(b)). The function *ObstacleDodge* uses a trivial algorithm to avoid an obstacle. The interesting thing in this function is the presence of synchronization mechanism (shown by the usage of fork and join states in the Real-Time Kernel partition).

The task *AdjustLeftPosition* (see Figure 15) uses a variable `sensLftStat` when calling the Real-Time Kernel service *SensorLeftRead*. Based on the value of that variable it decides whether to move forward, because the robot is correctly positioned on its left side; or to turn a little to the left. The word *little* might sound vague, but the numerical values shown in the Activity Diagram were actually calculated based on the speed of the robot and the width of the line. In the actual



(a) (b)

Fig. 14. Task ***CheckCollision***

implementation, the Lego Mindstorms robot slightly weaves in and out of the black line while moving forward.

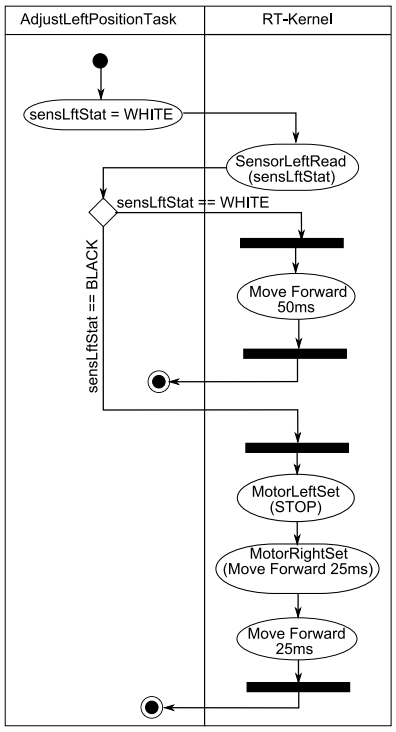


Fig. 15. Task ***AdjustLeftPosition***

V. Related Works

Several approaches and methods have been proposed to describe the process of building use cases. Although, none of them address the main topic of this paper, that is to elicit real-time tasks from user requirements, those previous works were very helpful to the development of this paper and served as a starting point to the further working out of several of the ideas here exposed.

Some of the first works on requirement engineering for real-time systems can be found in [18], [19]. Particularly, in [19] a formal model for analyzing requirements for real-time process-control systems is presented. In the case of [18], formal techniques for supporting the nature of real-time systems are developed. Some interesting results from the practical experience of applying use cases in the construction of real-time system are shown in [6]. The outstanding contribution of the previous paper had to do with the introduction of a *virtual actor*, which is a well-defined role played by an ill-defined user. Another report on practical experience, but focused on the automotive domain is [8]. On the other hand, [4] makes an extensive and deep analysis on the usage of use cases for real-time embedded systems by describing their suitability to this domain. In addition, the authors presented a method for guiding the process of requirements elicitation. Finally, in [7], the authors presented a method based on use case texts to derive objects in a real-time environment.

VI. Conclusions

Scheduling theory is the branch of real-time theory in charge of providing a designer with the necessary tools to make him sure that the system meets its intrinsic timing constraints. However, the theory is not very clear about how to define those tasks. In this paper, Requirements Engineering concepts were used to help the system analyst discover and characterize those tasks. Thus, making it easier to apply the real-time scheduling theory.

The analysis of any system begins with the gathering of user requirements. In the case of real-time systems, those requirements represents the functionalities of the system. This is, the tasks that compose the system. In this sense, a method for task elicitation from user requirements was developed using use cases. In addition, a generic use case architecture for real-time systems was also presented. This architecture is useful, since it includes most of the situations that can arise in this kind of systems and serves as a starting point for analyzing any system in this application domain.

Once tasks are elicited, the next step is describing their behavior. In order to do that, Activity Diagrams were restricted and adapted to real-time systems needs. Then, a method to turn use case texts into those restricted activity diagrams was presented. With this, each task behavior can be precisely described.

Finally, the proposed approach was empirically verified by analyzing a Line-Follower robot. The case study was implemented in a free real-time operating system and it was used to exemplify the concepts introduced in the paper.

References

[1] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok, "Real time scheduling theory: A historical perspective," Real-Time Syst., vol. 28, no. 2-3, pp. 101–155, 2004.

[2] M. Jackson, Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices. ACM Press, 1995.

[3] B. Nuseibeh and S. Easterbrook, "Requirements engineering: a roadmap," in ICSE '00: Proceedings of the Conference on The Future of Software Engineering. New York, NY, USA: ACM, 2000.

[4] E. Nasr, J. McDermid, and G. Bernat, "Eliciting and specifying requirements with use cases for embedded systems," in WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002). Washington, DC, USA: IEEE Computer Society, 2002.

[5] A. Cockburn, "Structuring use cases with goals," Journal of Object-Oriented Programming, Sep-Oct, Nov-Dec 1997.

[6] D. D. Zhang, "Use case modeling for real-time application," in WORDS '99: Proceedings of the Fourth International Workshop on Object-Oriented Real-Time Dependable Systems. Washington, DC, USA: IEEE Computer Society, 1999.

[7] M. Kimour and D. Meslati, "Deriving objects from use cases in real-time embedded systems," Information and Software Technology, vol. 47, no. 8, pp. 533 – 541, 2005.

[8] C. Denger, B. Paech, and S. Benz, "Guidelines - creating use cases for embedded systems," Frunhofer Institut Experimentelles Software Engineering, Tech. Rep., 2003, http://wwwagse.informatik.uni-kl.de/teaching/se1lab/ss2004/SysAnfBearbeiten/Guidelines_Short_Version.pdf Last visited: 12/09.

[9] "Usecases.org," http://www.usecases.org/ Last visited: 05/09.

[10] "Unified modeling language," http://www.uml.org.

[11] I. Jacobson and P.-W. Ng, Aspect-Oriented Software Development with Use Cases. Addison-Wesley Professional, 2004.

[12] M. Bakal, "Uml for c programmers," I-Logix, Tech. Rep., 2005, http://www.ddj.com/web-development/184401948 Last visited: 05/09.

[13] M. Bakal and J. Cohen, "Modeling c applications in uml with files and structures," Telelogic, Tech. Rep., 2008, http://www.dsp-fpga.com/articles/id/?3637.

[14] B. P. Douglass, "Uml for c programming language," IBM, White paper, 2008, http://download.telelogic.com/download/paper/RAW14058-USEN-00.pdf Last visited: 05/09.

[15] "Modeling and analysis of real-time and embedded systems," http://www.omgmarte.org/ Last visited: 05/09.

[16] H. Espinoza, J. Medina, H. Dubois, F. Terrier, and S. Gerard, "Towards a uml-based modeling standard for scedulability analisys of real-time systems," in International Workshop on Modeling and Analysis of Real-Time Embedded Systems at MODELS'06, Genova (Italy), October 2006.

[17] D. Donari and L. Ordinez, "Server oriented operating system," http://www.ingelec.uns.edu.ar/rts/soos Last Visited: 05/09.

[18] S. Goldsack and A. Finkelstein, "Requirements engineering for real-time systems," Software Engineering Journal, vol. 6, no. 3, pp. 101–115, May 1991.

[19] M. S. Jaffe, N. G. Leveson, M. P. E. Heimdahl, and B. E. Melhart, "Software requirements analysis for real-time process-control systems," IEEE Trans. Softw. Eng., vol. 17, no. 3, pp. 241–258, 1991.