# Eliciting and Specifying Requirements with Use Cases for Embedded Systems

Eman Nasr, John McDermid and Guillem Bernat
*Department of Computer Science, University of York, UK*
*{eman.nasr, john.mcdermid, Guillem.Bernat}@cs.york.ac.uk*

## Abstract

*This paper proposes enhancements to the use case technique for eliciting and specifying requirements for embedded systems. The work resulted from the employment of the use case technique for the requirements elicitation and specification of embedded systems in an industrial context. The use case technique is currently considered the state-of-the-art for handling requirements, because of the many benefits it provides. In spite of that, it still lacks proper definitions of the technique's constructs, and a well-defined process for practically applying the technique for requirements elicitation and specification. These are among the major issues that make the technique not readily suitable for employment for requirements elicitation and specification of embedded systems. This paper attempts to fill in this gap. The paper especially reports practical experience with a real-life case study from the avionics industry. It discusses the practical problems that were encountered and provides solutions.*

## 1. Introduction

The use case technique is currently considered the state-of-the-art for handling software requirements because of the many benefits it provides. It is widely acknowledged in the use case literature that among the important benefits of the use case technique are:

- Use cases elicit and specify requirements from the user's point of view.
- Use cases provide an excellent way for communicating with stakeholders.
- The process of use case modelling helps in bringing hidden requirements in the minds of the stakeholders to the surface where they can be specified.

In spite of that, the practical application of the use case technique to a real-life industrial case study for an embedded system turned out to be confusing for the following main reasons:

1- the use case technique lacks proper definitions of the technique's constructs to suit embedded systems,
2- the use case technique lacks a well defined process for practically applying the technique, and
3- the relation between requirements and use cases is not clear enough.

Although there is a considerable number of use case based approaches published in the literature, e.g. [1-7], nothing is readily suitable for the requirements elicitation and specification of embedded systems. This paper is an attempt to complement the literature by enhancing the definitions of the use case technique's constructs to better suit embedded systems, and by providing step-by-step guidance for the employment of use cases for requirements elicitation and specification.

The approach presented in this paper emerged as a result of experimenting with the use case technique for requirements elicitation and specification for an embedded system in an industrial context. The paper also reports on the practical experience of using the use case technique for a real-life case study from the avionics industry. It discusses some of the practical problems that were encountered and provides their suggested solutions. In addition, the paper discusses the new necessary concepts that were introduced to minimise the practical confusion while eliciting and specifying the requirements of an embedded system.

The paper is organised as follows. Section 2 gives an overview of the industrial real-life case study and some of its special properties. Section 3 discusses the main use case technique's modelling constructs in the light of their suitability to embedded systems, and proposes solutions for the confusing issues that we encountered during the course of the work. Section 4 presents step-by-step guidance for the employment of use cases for requirements elicitation and specification of embedded systems so as to minimise practical confusion. Finally, Section 5 concludes.

## 2. Overview of the industrial case study – Thrust Reverser Control System

This section gives an overview of the real-life industrial case study, which is from an aviation industry that specialises in the designed development of aeroplane engines. The aim from the work was to propose

Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)
1530-1443/02 $17.00 © 2002 IEEE

enhancements to the current software requirements engineering process. The case study is about an embedded control software that controls an aircraft's Thrust Reverser; namely, it is the Thrust Reverser Control System (TRCS). The Thrust Reverser is a device fitted in the exhaust system of an aeroplane engine that reverses the flow of the exhaust gases. It is used to assist in the deceleration of the Aeroplane and reduce the tear and wear of the brakes.

The Thrust Reverser hardware on the aeroplane's engine of the case study includes [8, 9]:

- Two pivoting blocker doors each activated by an hydraulic actuator.
- Four proximity sensors and two door position sensors to monitor the Thrust Reverser doors.
- Four lock latches controlled by two lock motors for opening and closing locks.

The TRCS is responsible for monitoring and controlling most of the Thrust Reverser hardware components, in addition to accepting commands and providing status information.

Among the properties of the TRCS that were noticed during the course of the study are:

1- The TRCS is deeply embedded in a highly complex engineered system; i.e. the aeroplane system.
2- The software is tightly coupled to the hardware; i.e. sensors and actuators, as it is responsible of controlling engineering activities.
3- The physical design of the TRCS hardware is already settled on.
4- Most of the functions of the TRCS are autonomous; i.e. activated internally by the system every specific time intervals.
5- Most of the external interactions of the TRCS are with other systems rather than humans.

These special properties of the TRCS, which are general to most embedded systems, pose special needs. The next section discusses the suitability of the use case technique's modelling constructs in the light of the above special properties, and offers solutions.

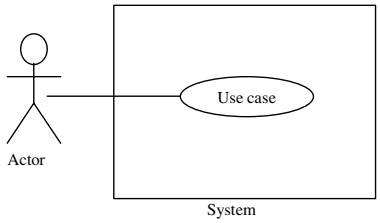


**Figure 1: The use case technique's main modelling constructs.**

## 3. Discussing the use case technique's modelling constructs

This section discusses the main use case modelling constructs as defined by Jacobson in [10] in the light of the case study's, and embedded systems', special properties, and offers solutions for the needs we encountered to avoid practical confusion. The use case technique has four main modelling constructs, as shown in Figure 1; *System*, *Actor*, *Use Case*, and *Communication Line*. Each of the four constructs is discussed in one of the following sub-sections.

### 3.1. The *System* construct

A *System* is modelled as a rectangle, as shown in Figure 1, with its name below. Although Jacobson has invented the *System* modelling notation for representing the system boundary, and to distinguish between what lies outside of the system under specification and what lies inside of it, he has not emphasised and discussed its pragmatics in his publications. In his publications he only discusses the pragmatics of the *Actor* and the *Use Case* constructs. We believe that this has led to finding, quite often in the literature, use case models with only *Actors* and *Use Cases* modelled without modelling the *System*.

We would like to emphasise the *System* construct and define what we mean by it, as this will have later implications on the rest of the use case modelling process. For systems that are composed of hardware and software, like that of our case study, and the domain of embedded systems in general, we find it more appropriate to consider the *System* to be the software and hardware composite. Because of the context of embedded software, which is usually deeply embedded in a larger highly complex engineered system, control software functions, which will be modelled as use cases, will usually control the different hardware components that are within the system boundary. It is difficult to separate software from hardware in embedded systems during modelling the requirements, as they are tightly coupled and highly interactive. Considering systems as hybrid, encapsulates the internal interaction between software and hardware within the system boundary, and, therefore, simplifies the use case model.

As soon as we reached such pragmatics for the *System* construct, our confusion about what to model as a system, the hardware or the software, was resolved. Therefore for the TRCS case study, we considered the system to be the software and hardware composite. The control software functions, which will be modelled as use cases, will control the Thrust Reverser hardware components, mentioned in Section 2, which are within the TRCS boundary.

### 3.2. The *Actor* construct

An *Actor*, modelled as a stick man in Figure 1 with its name below, represents a specific **role** played by an entity that resides **outside** the modelled system and interacts **directly** with it. We have written the three main concepts in the definition of an actor in bold, as they are crucial

keywords that could act as a rule of thumb for finding actors, which we shall deal with in more detail in Section 4.3. In the rest of this section we give a discussion of those three main concepts so as to eliminate practical confusion and reduce the time for identifying actors.

For the first concept: an actor represents a **role** of an external entity and not the external entity itself. This applies for any external entity and not only to human external entities. This is an issue that is overlooked by most of the use case researchers including Jacobson himself. That is why in Jacobson's famous Automatic Teller Machine (ATM) example, e.g. found in [10], he mistakenly models the "Bank System" entity, which is an external system to the ATM system (the *System* under concern) that needs to interact with it, as an actor. This is one of the major issues that cause practical confusion. Jacobson didn't stick to the definition he established for an actor. If the **role** of a human external entity is the thing that should be modelled in a use case model rather than modelling the person, and hence a person can play several roles towards the system under concern, then the same should be done for the external system entity. Hence in the ATM example the actor representing the "Bank System", should be one of the Bank System's **roles** towards the ATM instead, e.g. Account Handler. We would also like to emphasise that an external entity could be anything e.g. human or non-human (other systems, software or hardware, a thunderstorm, a bird, … etc.). This gives more flexibility to the pragmatics of the *Actor* construct. The use case literature often implies that what is meant by an external entity is either a human or another system only.

For the second concept: an actor lies **outside** the boundary of the modelled system. Therefore any entity inside the boundary of the modelled system should not be represented as an actor. Some use case literature, e.g. [1], suggest representing time as an actor for systems that involve functions activated in a certain interval of time. But we view time to be part of the system, i.e. within the boundary of the rectangular box, and we find it very confusing to suddenly decide to take part of the system outside of it, and model it external to the system when it is not. We understand that this is a way for representing an ill defined or a virtual actor, the terms introduced by Zhang [11], but we'd rather strictly stick to an actor's definition for two main reasons. First, as most of the functions for large complex embedded software are periodic, if we choose to have a time actor, then we have to model interaction between the time actor and all of the periodic use cases, which will make a use case model complex and less intuitive. Second, because we advocate considering a system to be the hardware and software composite, there will be other internal hardware sources of the system that could activate functionality, e.g. hardware errors. We also do not view the sensors that belong to a system under specification to be external entities; we view them to be internal sources for providing information.

For the third concept: an actor has to **directly** interact with the modelled system. Only actors that **directly** interact with the modelled system should be modelled in a use case model. Actors not interacting directly with the system should not be modelled, but if necessary, e.g. to help with the requirements elicitation phase, as we have encountered in our real life case study, Section 3.4 proposes a solution.

### 3.3. The *Use Case* construct

The functionality of a system is defined by different use cases, each of which represents a specific flow of events. The description of a *use case*, modelled as an oval shape in Figure 1 with its name inside, defines what happens in the system, and how the system interacts with the actors when the use case is performed.

Most of the use case literature emphasise that the use cases should be defined in terms of interactions between one or more external actors and the system to be developed. They propose what we call an actor-based strategy for identifying and defining use cases, as they identify use cases by focusing on the purposes of the actors and then define the interaction. However, not all use cases for all systems interact with external actors; there are systems that have significant functionality that is not a reaction to an external actor. Embedded control software systems provide a good example for such systems where major control functions are performed without significant external input. This makes the traditional use case technique seem less appropriate for such kinds of systems.

To offer a solution for such a limitation we would like to be more general and introduce the notion that a use case should focus on the purposes of the system as required by the stakeholders. We would like to bring to the surface and emphasise that use cases are defined according to the stakeholders' requirements in the first place. This way, the total collection of use cases will form really the complete functionality of the system under specification, whether they are associated with actors or not. In this way a use case could achieve a certain purpose for an actor of the system or achieve a certain purpose of the system being modelled. This results in the following changes in the use case pragmatics:

- A use case can be initiated internally by the system, e.g. according to time, and not only externally by an actor.
- A use case can describe internal functionality of a system and not only its external behaviour.

Use cases have been mostly used as a starting point for object-oriented analysis, design and implementation, although they are not object oriented. This makes the relationship between use cases and requirements fuzzy, as

it is very seldom explicit in the literature. This raises questions like: are use cases requirements? If not, how do they relate to them: do they originate from requirements? Or is it the other way round, do requirements originate from use cases? We would like to establish and emphasise the use case – requirements relationship, which is that use cases are repositories of requirements statements. A requirement statement could originate from stakeholders or from a previously written requirements document. The process we give in Section 4 will provide guidance on how to produce use cases that act as requirements specifications for the system under consideration.

### 3.4. The *Communication Line* construct

Like the *System* construct, the *Communication Line* construct has not been given enough attention in the traditional use case technique. The communication between an *actor* and a *use case* is modelled by using a solid line, as shown in Figure 1. The *Communication Line* links an actor to the relevant use case.

In the course of our work for the case study, we confronted situations where we needed to temporarily model indirect actors during the requirements elicitation activity until the direct actors were resolved. This is because there is usually some difficulty in identifying the direct actors for deeply embedded systems, as they often involve design decisions. We accounted for this by creating a new construct to represent an indirect communication link (see Figure 2) to be used only during the requirements elicitation activity. For example, in an Aeroplane system, the Pilot, as an actor, is responsible for controlling all of the Aeroplane hardware. But if we are concerned with modelling a system which is deeply embedded in the Aeroplane, like that of the TRCS, the Pilot will interact with this deeply embedded system through at least one other system in the Aeroplane. Figure 2 illustrates this example by the use of our new indirect communication link.

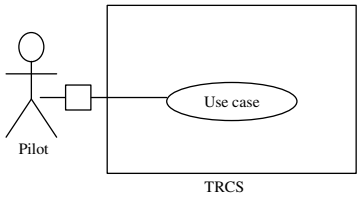


**Figure 2: Indirect interaction of Pilot and a use case of the TRCS.**

As shown in Figure 2, the indirect communication link notation is represented as a solid line going through a small box. The small box added on top of the normal communication line denotes a system, as most of the indirect interaction with deeply embedded systems happens through other systems.

## 4. Use case modelling process guidance

Most of the literature lack detailed step-by-step guidance for the use case technique. This section gives guidance for the use case modelling process. It is difficult to provide detailed guidance supported by examples from the industrial case study in this paper because of the size limitation. Therefore, we discuss the steps and how to achieve them using sub-steps, and only elaborate where we introduce new concepts or steps to the use case technique either to better suit the domain of embedded systems or to fix a limitation. The process steps are not meant to be strictly sequential, apart from the first step; they are only given for guidance.

Our process employs use cases for requirements elicitation and specification. Requirements elicitation is the activity through which the requirements of a system are discovered and elaborated through consultation with stakeholders, from previous documents, and from domain knowledge [12]. During the requirements elicitation activity the boundary for the proposed system is also defined. Ideally, whether a previous requirements document exists or not, we advocate holding requirements elicitation meetings that are facilitated by a Requirements Engineer. These meetings should include as many stakeholders as possible in order to provide coverage of all necessary requirements information. In each meeting, the meeting date and attendees should be registered as well, because this information will be needed in producing the final requirements documents so as to be able to trace the source of the requirements. In cases where a previous requirements document exists, like in the case of reengineering an already existing system, and the holding of requirements facilitation sessions are not feasible, the steps of the process should also be followed for elaborating and clarifying the requirements.

Each of the process steps is given in one of the following sub-sections. Steps 1 through 6 form the requirements elicitation activity, and Step 7 forms the requirements specification activity, which results in producing the requirements specification documents.

### 4.1. Defining the system

As we are employing use cases for the requirements elicitation activity, defining the system under specification is our first step in the process of use case modelling. The first thing we need to establish is what system we need to build. Most of the use case based approaches in the literature don't stress this first step. For example, in the Unified Software Development Process (USDP) [6], the first step is "Finding the Actors". We found that starting to define actors without first agreeing on a definition for the embedded system to be built resulted in identifying actors present in the application domain, in a wider context, whom might not need to communicate with the system under specification and thus

are out of our concern. That is why defining the system first is our strategy for narrowing down the scope. Besides, requirements cannot be effectively discussed at all without prior agreement on the which system one is talking about and at what level of abstraction. From our experience defining the system is a crucial step to avoid practical confusion and save requirements modelling time. To elaborate more on how to achieve defining the system step we give the following three sub-steps:

a) **Naming the system.** A name should be given to the system under specification that makes sense to all of the stakeholders of the system.
b) **Eliciting system rationale.** The system's rationale is a very important piece of information which is usually overlooked. Identifying the rationale of the system will add to the depth of understanding.
c) **Writing a brief description of the system.** This step is to define the system generally, identify the overall goals, and document the general features wanted. The brief description of the system will serve as the problem statement, and provide the starting point for eliciting the requirements of the system.

   For a hybrid system, like that of the TRCS case study, if the system under specification is a low level system, where the physical components of the system are already settled on, the system hardware components should also be included in the system description together with a physical context diagram. This is because the hardware is taken to be part of the low level system, as explained before, and the control requirements of the system will need to make explicit reference to its components.

### 4.2. Defining developmental quality requirements

Most of the use case based approaches in the literature only concentrate on the behavioural requirements of the system under consideration, and there is little guidance, if any, on how to deal with the developmental quality requirements. We introduce a 'Defining Developmental Quality Requirements' step to overcome the limitation identified in the current use case based approaches. This step is to define the overall static requirements for the system; i.e. the constraints placed on the whole system. This includes requirements such as the desired programming language for a software implementation, software platform constraints, hardware design constraints, the cost of the system, and regulatory requirements. Defining the developmental quality requirements of the system under consideration, involves applying the following two sub-steps:

a) **Eliciting the system's developmental quality requirements.** A list should be created with all of the developmental quality requirements for the system under consideration. We used structured English "shall" statements to specify the developmental quality requirements. For example, to specify the desired programming language for a software implementation, the statement looked as follows: 'The system shall be implemented using Ada.' Composing questions as the following and looking for their answers provide some guidelines for identifying the developmental quality requirements:
   i) Are there any standards that need to be followed?
   ii) Are there any safety requirements for the system?
   iii) Are there any requirements for the desired programming language?
b) **Numbering the developmental quality requirements sentences.** Each sentence of the developmental quality requirements should be given a unique number for traceability purposes.

### 4.3. Finding actors

Finding the actors of the system is a step that is common to all of the use case based approaches in the literature that provide a process, e.g. USDP [6]. In fact, it is even considered to be the first step in most of their processes, but there is very little guidance on how to achieve this step. We provide guidance to overcome such a limitation by recommending the following three sub-steps:

a) **Identifying the external entities that need to interact with the system.** By first identifying the external entities that need to interact directly with the system under specification, then analysing their roles towards it, a candidate list of actors can be defined. Identifying roles played by human entities towards the system under specification are much easier than identifying roles played by non-human entities. This is because the roles of the human entities are often reflected in the job/position title, which usually has a clear definition and a set of defined responsibilities [11], e.g. the Pilot and Maintainer in an Aeroplane system. Therefore, in order to define a role for another system entity (or any other non-human entity) that needs to interact directly with the system under specification, it should be clearly defined and its responsibilities towards the system under specification analysed. By identifying the responsibilities, roles (i.e. actors) could be identified. By the use of this rule of thumb the confusion about identifying actors for non-human external entities will be eliminated.
b) **Naming each actor.** Each potential actor identified should be given a name that makes sense to all of the stakeholders of the system.
c) **Describing briefly each actor.** Creating a brief description, which defines the actor and its responsibility towards the system under specification. This proved to be helpful in linking the actor to the external entity that implements it.

### 4.4. Finding use cases

Finding the use cases of the system is also a step that is common to all of the use case based approaches in the literature that provide a process. However, most of the use case based approaches have only one strategy for finding use cases, which is actor-based. In our approach we don't only depend on the actor-based strategy to identify use cases. Our main strategy for finding use cases is a system function-based one to offer more flexibility and ability to identifying all of the functional use cases of a system under specification. We detail the following major three sub-steps to achieve 'finding use cases' step:

**a) Identifying the major functionality of the system.** Our strategy is to initially identify the major functionality of the system, as use cases are what the system does. As we mentioned before, the traditional way in the use case based approaches to identify use cases is to link them with the actors of the system by identifying use cases for each actor found for the system. Although this provides very useful guidance, in the domain of embedded control software systems, a major responsibility lies on the domain specialists to identify what the system should do. This is because of the nature of the domain; major autonomous use cases would need to be performed by the system without the need to interact with an external actor.

During the course of the TRCS case study, we found that there are three major types of use case functionality for an embedded control software system, which are for:

- initialisation of the system;
- monitoring specific sensors, which results in controlling specific actuators within the same use case; and
- reporting status information.

It is also true that the system also accepts external commands and reacts to them, but we don't consider this to provide with extra use case types as those situations are usually documented within one of the sequence of events of the use case types mentioned above. We believe that the above three types of use cases would form the basis of any embedded control software system functionality. Indeed there might be other types of functionality which we didn't come across in the course of our study, and that would need to be further investigated to be able to provide a comprehensive list of all types of possible use cases for a software control system. We only mean that this experience could be used for guidance so as not to miss an important use case of the control software system under specification.

The following questions could be used as guidelines for identifying use cases:

i) What are the primary tasks that the system should perform?
ii) Will the system need to perform any monitoring functions?
iii) Will the system need to perform any control functions?
iv) Will the system need to perform any initialisation functions?
v) Will an actor need to create, store, change, remove or read data in the system?
vi) Will an actor need to inform the system about external information?
vii) Does an actor need to be informed about certain occurrences in the system?

The above questions will help in generating a candidate list of use cases for the system under specification. From our experience with the real life case study, we advise that care should be taken to only identify what is needed from the system under specification. In the domain of embedded control software systems it is challenging to only describe the functionality of the system under specification because of the high interactiveness between it and other embedded systems in the complex engineered system. The definition of the system under specification identified before should be always used as a guide to help in limiting the scope.

**b) Naming each use case.** Choosing a proper use case name is one of the confusing issues in the current use case based approaches. Although not mentioned explicitly in the literature, the traditional strategy for naming use cases is to choose an active verb phrase that indicates the purpose of the actor. As reported by practitioners in [11], a possible ambiguity could arise in naming, as sometimes a use case is named after the behaviour of the actor instead of the system. A good example for this ambiguity is Jacobson's naming one of the ATM system's use cases "Withdraw Money," which is a function to be done by the "Client" actor and not by the system. To eliminate such an ambiguity we adopted Zhang's rule [11] and adapted it to account for use cases that are triggered internally by the system to form the following rule to help with the naming convention of a use case:

*The <system name> is requested [by the <actor name>] to perform* ***<use case name>****.*

In the above rule, the phrase section between the square brackets indicates that it is optional, as according to our approach a use case could be triggered by an actor or could be autonomous; i.e. triggered by the system, as we previously emphasised in Section 3.3. In the above rule the <system name>, and the <actor name> (if applicable), are to be replaced by the name of the system under specification, and the name of an actor respectively, so as to be able to reach a suitable **<use case name>**. In this way the use case name reached will indicate what

the system under specification is to perform, and not the actor. To give an example, we apply the above rule to find a better name for the "Withdraw Money" use case in Jacobson's ATM example to get:

*The ATM System is requested by the Client to perform **Dispense Money**.*

An a ATM System's function is not to withdraw money, but to dispense money instead.

A person might argue that it is obvious that the one who will withdraw the money is the Client actor and not the ATM system. To such an argument we reply that this is obvious as this particular example is simple and easy to envision, as the Client actor plays a role implemented by a human entity. But when the system gets complex, and the actors are implemented by non-human entities, e.g. other interacting systems, there has to be an unambiguous way for naming the use case to obviously specify what the system under consideration is requested to perform rather than the external entity.

c) **Describing briefly each use case.** Creating a brief description for each use case helps to define each use case's scope. The scope should indicate the area of functionality that the use case will describe.

### 4.5. Creating the use case model

There is nothing different in creating the use case model. It is a step common to all use case based approaches in the literature.

### 4.6. Describing the use cases

For each identified use case, use case sequence of events information should be elicited. The requirements found in the set of use cases comprise the set of requirements for the required system. For each use case that has been identified the following four major sub-steps should occur; this is an iterative process of progressive refinement, which should be performed until the involved stakeholders feel that all use cases have been described satisfactorily:

a) **Elaborating the use case brief description.** For each use case, the use case's brief description previously written should be elaborated. The purpose is to provide with a structure around which the sequence of events could be based.

b) **Defining the use case's flow of events.** This step should be done with the help of stakeholders in order to provide coverage of all of the necessary information for the use case. First, from the use case's Brief Description, the detailed flow of events of the use case could be defined. Extending the Brief Description to define the flow of events will ensure that the flow of events is within the scope of the use case. Each single step in the sequence of events forms a requirement statement that we wrote on a separate line using structured English shall statements. The flow of events should include a description of pre-conditions, the main flow of events, how and when the use case begins and ends, when the use case will interact with actors and what is exchanged between them, what the system needs to do to perform a use case, how and when the use case will need data stored in the system or will store data in the system, exceptional events, and post-conditions.

The initiation of a use case occurs whenever the pre-conditions are met, and the starting event occurs. Pre-conditions must always hold through the execution of a use case; i.e. a violation of the pre-conditions is not to be considered within the related use case. This is one of the very confusing issues about preconditions that is not well defined in the literature.

c) **Numbering the use case's Flow of Events.** After listing the use case's flow of events, each individual sentence (requirement) should be uniquely identified and numbered for requirements management and traceability purposes.

d) **Identifying the use case's unresolved issues.** The unresolved issues of the use case should also be documented. The purpose is to identify the existing requirements that need greater clarification. These identified use case unresolved issues should be listed in an issues section within a use case document.

**<System Name>**
**Overview Document**

1. History
   - 1.1 Author
     Identifies the author of the document.
   - 1.2. Document Description
     Identifies the purpose of the document and lists its contents.
   - 1.3. Document Version
     Contains document version number
   - 1.4. Date of Document
     Contains the document creation date.
   - 1.5. Date of Elicitation
     Contains the date of the elicitation session(s) held for the purpose of creating this Overview Document.
   - 1.6. Stakeholders and Roles
     Identifies the names of the stakeholders sharing in the elicitation session(s) for this System Overview document and their respective roles within the elicitation session(s).
   - 1.7. Other Sources of Information
2. System Rationale
   Contains the rationale of building the system.
3. Description of the System
4. Names and Descriptions of the Actors
   Contains names and brief descriptions of the actors involved in system.
5. Candidate Use Cases
   Lists the names of the candidate use cases.
6. Use Case Diagram
   Contains the use case diagram of the system.
7. Developmental Quality Requirements
   Lists any developmental quality requirements for the system.
8. Abbreviations/Acronyms/List of Terms
9. Issues
   Contains issues identified that need to be resolved by the stakeholders.

**Figure 3: System Overview Document template.**

### 4.7. Authoring documents

We have two main documents for documenting the requirements that result from the requirements elicitation activity. The general system information is to be recorded in a System Overview Document, named after the name of the system, while the specific use case information is to be documented in an Use Case Document, named after the name of the use case.

a) **Authoring the System Overview Document.** This document is important in that it gives context and serves to define the specified system. It forms the links between all the Use Case Documents and contains how the use cases and actors fit together in the form of a use case model. Figure 3 gives a general template for what the System Overview Document may contain.

b) **Authoring Use Case Documents.** For each use case defined and described an Use Case Document should be authored, which will be initially used for further validation by the stakeholders. Figure 4 gives a general template for what the Use Case Document may contain.

**<Use Case Name>**
**Use Case Document**

1. History
   - 1.1. Author
   - 1.2. Document Description
   - 1.3. Document Version
   - 1.4. Date of Document
   - 1.5. Date of Elicitation
   - 1.6. Stakeholders and Roles
   - 1.7. Other Sources of Information
2. Rationale
3. Participating Actors
4. Brief Description
5. Flow of Events
   - 5.1 Pre-Conditions
   - 5.2 Main Flow of events
     - 5.2.1 Normal Sequence
     - 5.2.2 Alternative Events
     - 5.2.3 Exceptional Events
   - 5.3 Post-Conditions
6. Associations to other use cases
7. References to other documents
8. Traceability
9. Abbreviations/Acronyms/List of Terms
10. Issues

**Figure 4: Global Use Case Document Template.**

## 5. Conclusions

This paper has presented enhancements for the use case technique to better suit embedded systems. In the light of an embedded control software system case study from the aviation industry, the paper has provided discussions of, and enhancements to, the main use case constructs, in addition to providing step-by-step process guidance for ease of practical application. We have attempted to elaborate the relationship between requirements and use cases, and employed use cases for requirements elicitation and specification. Employing the enhanced technique to the industrial case study offered better understanding of the system, its environment, and its workings; assistance in discovering missing requirements information; better definition of system boundary, which is a significant problem for requirements engineering in the domain; controlling the repetition of requirements, within and across system requirements documents; and capturing behavioural requirements, where functional requirements are linked to their respective non-functional ones, e.g. timing requirements. Although our approach emerged from the embedded systems domain we contend that they are of wider applicability and that it improves the practicality of the use case technique in general.

## Acknowledgments

This work has been funded by the European Commission [PRECEPT (Promoting Requirements Engineering from Current Engineering PracTices) project 25412]. We would like to acknowledge the input of the PRECEPT partner, Phillips Research Laboratories, Surrey, to the use case modelling process guidelines.

## References

[1] Schneider, G. and J. P. Winters. *Applying Use Cases: A Practical Guide*. Addison-Wesley, 1998.

[2] Jacobson, I., M. Christerson, P. Jonsson and G. Oevergaard. *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[3] Jacobson, I., M. Ericsson and A. Jacobson. *The Object Advantage: Business Process Reengineering With Object Technology*. ACM Press, 1995.

[4] Douglass, B. P. *Real-Time UML: Developing Efficient Objects For Embedded Systems*. Addison-Wesley, 1998.

[5] Awad, M., J. Kuusela and J. Ziegler. *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion*. Prentice-Hall, 1996.

[6] Jacobson, I., G. Booch and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[7] Robertson, S. and J. Robertson. *Mastering the Requirements Process*. Addison Wesley, 1999.

[8] *A330 Flight Crew Operating Manual*.

[9] *Control System Airframe Interface Document for the BR700-715 C1-30 MD95 Application*. E-TR 386/96-(IR)-ISS03. BMW Rolls-Royce AeroEngines, 1997.

[10] Jacobson, I. The Use-Case Construct in Object-Oriented Software Engineering. In J.M. Carroll, (ed.) *Scenario-Based Design: Envisioning Work and Technology in System Development*, pp. 309-336, John Wiley and Sons, 1995.

[11] Zhang, D. D. Use Case Modeling for Real-Time Application. In *Proceedings of The Fourth International Workshop on Object-Oriented Real-Time Dependable Systems*, pp. 56-64. IEEE Computer Society, 1999.

[12] Loucopoulos, P. and V. Karakostas. *System Requirements Engineering*. McGraw Hill, 1995.