Deriving objects from use cases in real-time embedded systems

Mohamed T. Kimour*, Djamel Meslati

Department of Computer Science, Bp. 12, University of Annaba, Annaba, Algeria

Received 25 December 2003
Available online 8 December 2004

Abstract

In recent years, a number of use case-driven processes have emerged for the development of real-time embedded systems. In these processes, once requirements have been defined by use cases, the next step is usually to identify from that use cases, the central objects in the system and describing how they interact with one another. However, identifying objects/classes from the requirements is both a critical and hard task. This is mainly due to the lack of pragmatic technique that steers such a task. In this article, we present a systematic approach to identify objects from the use case model for the real-time embedded systems. After hierarchically decomposing the system into its parts, we first transform the use case structured-text style into an activity diagram, which may be reused in the next development activities. Second, we use the derived activity diagram for identifying objects. With the behavioural model, an object model can be viewed as a first cut at a design model, and is thus an essential input when the system is shaped in design and design implementation.

Keywords: Real-time embedded system; UML; Use case; Activity diagram; Object model

1. Introduction

Real-Time embedded systems are becoming increasingly sophisticated and complex, while at the same time experiencing a shorter time to market with greater demands on reliability and security.

As a result, the need for systematic software development methods and efficient tools for real-time embedded systems is now greater than ever [25]. UML [7,24] is currently the most widespread software modeling language. The emergence of UML as an industry standard for modeling systems has encouraged the use of automated software tools that facilitate the development process from analysis through coding. The most notable UML-based approaches for real-time system development are based on use cases to capture requirements [3–6,20,21].

Use cases describe interactions between the system and its environment and thus capture the functional system requirements. In the system development process, static and dynamic models of the system are built. The static model defines the structural relationships among domain classes, which are covered by the UML object/class diagrams [7]. Object/class model is used as the foundation for the design and implementation phases [8,9].

Once the requirements have been specified by a means of use cases, the developer identifies the objects and classes that describe the system under development [1]. Behavioral diagrams such as interaction model cannot be constructed without knowing the concerned objects [10]. This is why it is fundamental to precisely identify these objects early in the system development process.

Nowadays, there are many methods that treat the problem of objects and classes identification [5,6,8,9,11,12,20,21,23]. Unfortunately, these methods suffer from at least two problems: first is a focus on classes rather than objects and, second, a direct identification of classes and/or objects from the use case textual description.

However, when developing a real-time embedded system, it is easier and more important to start with building the object model instead of the class model [8,13]. Class models are inappropriate when more than one object of the same class is used in a specific situation. This is the case in the real-time embedded systems where the elements that do constitute these systems are concrete entities that can be
directly mapped to objects. In contrast, classes are templates that are used for behavioral sharing purpose and do not correspond to concrete elements. Moreover, classes’ emphasis is put on commonalities, whereas, objects in the embedded systems are often specific.

On the other hand, transition from use cases to objects is not straightforward, because there is no direct one-to-one mapping between these two models. Moreover, identification of objects depends on the use cases representation. Impreciseness, ambiguities, and inconsistency may be present in the use case text. These drawbacks make the object identification process difficult and usually resulting in unsatisfactory object or class model.

Current methods that are dedicated to object and/or class identification can be classified into two categories. In the first one, the methods are based on linguistic analysis of the requirements document which is written in a natural language. Objects and attributes correspond to the nouns, and the operations are related to verbs. In the second category, some tools that help automate the transition from use cases to class and object diagrams are proposed. However, they only can treat a very restricted form of use cases. For both, the main problem is the vagueness, ambiguity and inconsistency of the natural language.

In this article, we propose an approach to object identification from use cases using activity diagrams. First, after a hierarchical decomposition of the real-time embedded system into its controlled parts and controlling subsystem, we convert the use case text into an activity diagram, where activities, actions and events refer to the system controlled parts. During this conversion process, detected ambiguities, inconsistency and impreciseness are removed.

Second, we identify objects that directly concern each controlled unit and those of other types according to their categorization and the separation of concerns principle [14,15]. Once objects have been identified, classes to which those objects belong can be determined later. In this way, our approach is consistent with the Rumbaugh’s bottom–up method to discover inheritance links and organize classes [16].

Using activity diagrams as a bridge between use cases and object model, our approach not only allows object identification from use cases in a systematic manner, but also provides the benefit of an improved requirements specification. The identified objects, in particular those that are related to the domain entities allow for enriching use cases and making them clearer, more adequate and useful.

The rest of the article is organized as follows: Section 2 overviews the main activities of a real-time embedded system development according to the unified software development process, and describes UML models used in our approach, namely, use cases, activity diagrams, and the object model with the object categorization. Section 3 presents our object identification process. In Section 4, we provide an overview of the related works and compare them with our approach. We will conclude our work and outline future directions of our research Section 5.

2. Real-time embedded system development

In recent years object-oriented techniques have been used in the development of real-time embedded systems. Examples are ROOM [26], RT-UML [5], UML-RT [3], COMET [6], OCTOPUS [21], etc. In these methods, the development flow is usually composed of requirements analysis, design, implementation, and test phases.

The first step in the analysis phase covers the definition of the system requirements and the extraction of functional and non-functional properties, from the problem description. The second step in this phase defines the system structure (object/class diagrams) and behavior (interaction diagrams, state machines, and activity diagrams). Object/class identification is an important step in the analysis phase, in the sense that object/class models are used as inputs to the subsequent phases such as design and implementation.

The second phase is the design one, which is the process of specifying an implementation that is consistent with the results of the analysis. It consists of a refinement of the models defined during the analysis phase and of an explicit specification of the communication, synchronization, and data exchange between objects. Moreover, the evaluation of the design models is a very important task in this phase, in the sense that it allows for detecting errors before starting the implementation. Implementation and test are the two last phases in these development processes.

These above-mentioned methodologies use an extension of UML in order to specify the requirements (reliability, safety, timeliness) needed for a successful development of real-time embedded systems.

In these systems, software controls a small portion of the world [17,18]. It interacts with its environment. It monitors environmental properties and introduces changes through modification of the logical values or physical actuators that it controls. From this simple observation, we group the activities of these systems into three classes: (1) monitoring activities; (2) computation activities; and (3) controlling activities.

We will redefine the object categories based on this activity classification. Object categorization allows for defining objects to encapsulate portions of the behavior specified by activity diagram.

To provide more clarity and preciseness to the use case text, we start with a hierarchical decomposition of the system. Thanks to this decomposition, activities, actions and events will be related to the controlling units of the system, such as a door, a cabin, and a floor, in an elevator control system.

In addition to the description of the object categories that are relevant to real-time embedded systems, we present, in the following, the UML 2.0 models that we use in our object
identification process, namely, use cases and activity diagrams.

2.1. Use cases

A use case is a specific way of using the system by performing some part of the functionality [20]. In real-time embedded systems, an actor can be a human being, a computer system, an external I/O device, or a timer. External I/O devices and timer actors are particularly prevalent in these systems [6]. A complete set of use cases specifies all the different ways to use the system, and therefore defines all behavior required of the system.

As use cases serve as a means of communication between developers and users, they are fundamentally written in simple text. However, their textual description presents some drawbacks such as the lack of precision and conciseness.

In our object identification process, we are interested in the description of a use case defined by a name, actor, preconditions, postconditions, normal steps, and alternative steps according to Cockburn’s template [19]. To this template, we have added a quality of service section in which we describe non-functional requirements (response time, security, cost, accuracy, etc.).

Fig. 1 illustrates a typical textual description of a request elevator use case in an elevator control system. A use case can be seen as a tuple \( <\text{ucName}, \text{ucActor}, \text{ucPre}, \text{ucPost}, \text{ucSteps}, \text{ucAlt}, \text{ucQoS}> \).

\( \text{ucName} \) is a label that uniquely identifies a use case, \( \text{ucActor} \) is a primary actor and the secondary actors, \( \text{ucPre} \) is a set of preconditions, \( \text{ucPost} \) is a set of postconditions, \( \text{ucSteps} \) is a set of ordered normal steps, and \( \text{ucAlt} \) a is set of alternative steps, and \( \text{ucQoS} \) is a set of qualities of services.

Each step in \( \text{ucSteps} \) is a tuple \( <sNumber, sAction> \) with \( sNumber \) a step number, \( sAction \) a set of actions (actor action(s) or system response(s)). An action may also be a branching statement to another step. A normal step may be associated with a set of alternative steps.

An alternative step can be seen as a tuple \( <\text{altStepNumber}, \text{guardCond}, \text{altStepAction}> \), with \( \text{altStepNumber} \) an alternative step number and \( \text{guardCond} \) a guard condition on this step, and \( \text{altStepAction} \) an alternative set of actions. A subset of use case steps in an automated teller machine system may be as follows:

\[
\{<1, \text{User inserts card}>, <2, \text{System Asks for PIN}>, <2a, \text{[invalid card]}>, <2a1, \text{System emits alarm}>, <2a2, \text{System ejects card}>\}.
\]

2.2. UML 2.0 activity diagrams

In this section, we introduce the UML 2.0 activity diagram concepts that we use to model system level behavior of use cases. UML 2.0 [24] provides activity diagrams with better constructs (Fig. 2), making them more effective and flexible in describing use cases.

They have recently undergone a major revision and redefinition of important concepts like, activities, actions, control and data flows, concurrency, procedure call, and exception handling that are very useful in modeling real-time embedded systems. Moreover, these diagrams are a graphical technique that provides a relatively simple and abstract representation using easy-to-learn notation. The obvious advantage of this is that they offer a means of communication between developers and clients, and a valuable tool for requirements elicitation.
An activity describes a logical unit of work. It can be broken down into actions. An action is the smallest unit of work that is not decomposed any further. The sequencing of actions or activities is controlled by control and object flow edges. There are three kinds of nodes: activity node, object node, and control node. An object flow is an edge that can have objects or data passing along it. It models the flow of values to or from object nodes.

Activities may contain actions of various kinds:

- Occurrences of primitive functions, such as arithmetic functions,
- Invocations of behavior, such as activities,
- Communication actions, such as sending/receiving signals,
- Manipulations of objects, such as reading or writing attributes or associations.

Activities/actions are joined by edges that represent process flows or events. A decision node can model divergent behavior based on a condition. Synchronization points may also be defined to illustrate how processing may be carried out in parallel, then synchronized at a point before further activity is undertaken. Input and output parameters can be shown in an activity node. This is done via rectangles that are attached to the activities.

2.3. Object modeling and categorization

The UML object diagram is useful for exploring real world examples of objects and the relationships between them. This diagram provides a conceptual description of the entities in the application domain. It complements use cases in describing requirements and provides an initial architecture that first captures these requirements. It is an important model being used at almost all the steps of the system development activities.

At the requirements analysis level, objects are usually determined from use cases according to certain object categorization [6,8,20]. Jacobson et al. [20] have divided the analysis space in three orthogonal dimensions: information, behavior and presentation. We believe that object categorization makes models more stable and comprehensive.

For the purpose of identifying objects from use cases in real-time embedded systems, we redefine the object categories as follows:

- **Coordination object.** It is an overall decision-making object that determines the overall sequencing for a collection of related objects inherent to a use case. It does not encapsulate any computation other than that is needed for the coordination. Its main responsibility is to supervise the other objects in order to achieve the use case goal.
- **Interface object.** It handles the exchange between the system and its environment. An interface object must be encapsulated in a way such that if a change is made to the exchange between the system and its environment, only the interface object has to be modified, leaving unchanged the other objects.
- **Entity object.** Long-living object that stores information (typically the entities in entity-relationship models). The entire behavior associated to the manipulation of that information must be included in the entity object. An entity object is typically accessed by many use cases.
- **Application logic object.** It contains the details of the application logic. It is needed to hide the application logic separately from the data being manipulated (because it is likely that the application logic could change independently of the data).
- **Control object.** It encapsulates appropriate computations that involve a group of entity objects or that cannot be naturally associated to other objects.
- **Timer object.** It encapsulates the temporal conditions and triggers activities in other objects periodically or at appropriate time points.

3. Object identification process

In this section, we present our object identification process and show how it can be integrated in the use case driven activities for developing real-time embedded systems according to the Unified Software Development Process [4].

By building activity diagrams from use cases, we begin the analysis phase by the dynamic model instead of the static one as it is done in almost all the current real-time embedded system development techniques. If one begins with the dynamic model, it is easier to identify actions, events and also attributes and group them into object categories.

Fig. 3 depicts the main activities leading to the object identification. First, we have to understand and describe the most important concepts of the system to be developed. A system context model is built to depict the interactions.
between the system and its environment. Afterward, the interactions identified between actors and the system should be identified by means of use cases. Each use case should be specified in a structured text style. For more comprehension of the use cases and easily building activity diagrams, a hierarchical decomposition of the system should be performed.

Afterward, we derive from each use case an activity diagram in an iterative and incremental manner (main feature of the unified software development process) according to an appropriate derivation process that we describe in Section 3.1.

During this step, the use case text may be reviewed, modified and enriched to exhibit more preciseness and completeness. Moreover, detected ambiguities, inconsistencies, and errors are removed. Based on the derived activity diagram, objects will be identified according to an appropriate identification process that we describe in Section 3.2. Identified objects and their relationships will constitute the analysis object model of the system.

3.1. From use cases to activity diagrams

Table 1 describes our procedure to transform a use case text into an activity diagram. This procedure is structured into the following iterations:

**Activity Iteration.** The use case text is transformed into an activity diagram (Fig. 4). In this iteration, we specify possible concurrency between activities and place guard conditions on edges or within the decision node.

**Action Iteration.** We detail each activity by specifying the corresponding actor’s actions or system’s responses according to the general format: ‘actId.actionDes’, with actId an actor identifier to distinguish between actor’s action and system’s response, and actionDes the action’s name. An action should refer to no more than one system unit or one actor.

**Consistency Iteration.** We check the activity diagram for internal completeness and consistency, and identify the possible timing constraints.

In the following, we discuss step by step, how activities and actions are determined from each use case’s step and, using our use case example, we show how to build the activity diagram. Fig. 5 depicts the activity diagram derived after applying the three above-mentioned iterations.

**Step 1.** Two actions are specified in this step: (1) user presses an up floor button, and (2) system selects an elevator. This last action specifies that the system should select an elevator to visit the source floor. This description is ambiguous, in the sense that it isn’t precisely defined. Should the system determine any elevator, or should it determine the most suitable one? To determine the most suitable elevator in a multiple-elevators control system, data items such as the source floor number and the desired direction should be available. In addition, the controlling subsystem should get access to data items specifying the status of each elevator (idle, moving up, moving down, last visited floor).

Table 1

<table>
<thead>
<tr>
<th>Activity diagram derivation procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Activity iteration</strong></td>
</tr>
<tr>
<td>For each use case, build an activity diagram, where, each node corresponds to a step, and each edge links two nodes that correspond to the two consecutive steps in the use case. A node becomes an activity with a name. Model the normal flow first, integrate the alternative flows later. Specify possible concurrency between activities. Specify event names and possible guard conditions on the corresponding edges.</td>
</tr>
<tr>
<td><strong>Action iteration</strong></td>
</tr>
<tr>
<td>Decompose each activity into actions. An action should refer to no more than one system unit or actor. Specify the order and possible concurrency between actions. For each action, specify the necessary accessed data items. Specify event names and possible guards on the corresponding edges between action nodes.</td>
</tr>
<tr>
<td><strong>Consistency iteration</strong></td>
</tr>
<tr>
<td>Check the activity diagram to achieve: Activities, actions, guards, events, should be named expressively and consistently. All the necessary activities, actions, and events should be specified. All the activities in an activity diagram should be connected. Check the event list created to see if all relevant events are handled, and if all the necessary activities and actions are specified. Identify the possible timing constraints and place them on the corresponding edges.</td>
</tr>
</tbody>
</table>
Therefore, the first two data items should be specified. Thus, during the activity diagram elaboration we uncover the ambiguity related to the described action ‘selecting an elevator’ and provide the necessary precision by determining all the necessary events and data items.

**Step 2.** In the node of the corresponding step, we specify the action ‘S.determineDirection’.

**Step 3.** Two actions are explicitly specified in this step: ‘closeDoor’ and ‘startMoving’ that correspond to the system commands of closing the door and start moving elevator. However, commanding the elevator to start moving should be followed by updating its status. Therefore, the missing action ‘updateStatus’ should be added to the corresponding node of this step. In doing so, we uncover an omission in the use case. In addition, for the sake of security, the elevator must start moving at the earliest at certain time units after the door has been closed. This important timing constraint is not explicitly specified in this use case. In doing so, we uncover a possible temporal inconsistency.

**Step 4.** The secondary actor ‘floor sensor’ detects the fact that the elevator is approaching a floor that becomes current, and notifies the system. The system updates the current floor information of the appropriate elevator.

**Step 5.** The system checks if the current floor is in the list of the floors to visit by the concerned elevator. However, this action needs accessing to all the pending requests, including the one that is submitted by the user at the first step. When examining the previous steps, we do not find any reference to adding this request. To uncover this omission in the use case, we add the action ‘addRequest’ in the node, after the action ‘determineElevator’.

**Step 6.** The system commands the door to open. We therefore place the action ‘openDoor’ in the corresponding node.

**Step 7.** The system checks if there are other requests. If so, the door must wait for a certain moment before to close.

This temporal requirement is not explicitly defined in the use case. Therefore, we explicitly specify this timing...
constraint as a label on the edge that starts from the door opening node and ends at the direction-determining node. If there are no other requests, the elevator stays at the current floor with the door open.

However, the information status of this elevator that becomes idle must be updated. We therefore specify the action ‘updateStatus(idle)’ in the corresponding node.

3.2. From activity diagrams to objects

The derived activity diagram, which models activities on the system’s controlled units, incorporates the necessary elements to define objects such as, actions and data items. For each controlled unit, we determine the necessary objects according to the procedure defined in Table 2.

Fig. 6 depicts the object model corresponding to the elevator request use case example, obtained by this derivation process. In the following we show how the object model is derived.

The controlled units appeared after a hierarchical decomposition of the system highlighted by the derived activity diagram, are a set of elevators, a set of elevator doors, and the floor sensors.

For the sake of brevity, we do not consider other units such as buttons and lamps. In the following, we discuss the objects identification from the activity diagram:

1. We define for the whole use case, a coordination object (named elevRequestCoordinator). This object does not encapsulate any functional action. It only triggers actions encapsulated in other objects or receives signals from them.

2. For each controlled unit and actor that receives and/or sends events from/to the system, we define an interface object to receive inputs from these units or actor, and/or to send outputs or commands to them. In our example, we define interface objects for the identified controlled units (elevators, doors, and the floor sensors).

For instance, we define, for every elevator, an interface object ‘elevatorInterface’ to receive commands (startMoving(), stopElevator()) from the system. Also, for every elevator door, we define an interface object ‘doorInterface’ to receive commands (openDoor, closeDoor), and for the floor sensor, we define the interface object ‘floorSensorInterface’, to handle interaction at every elevator approaching a floor.

3. Actions such as addRequest(), and updateStatus() refer to an elevator. We therefore define an entity object, named ‘elevStatusPlan’, for each elevator.

Moreover, from the guard condition ‘elevator idle’, we identify the attribute ‘elevator status’, which may have the value ‘idle’. This attribute should also be encapsulated by the entity object ‘elevStatusPlan’.

This entity object should have information on whether an elevator is moving or idle, as well as on the current floor if it is at a floor or the last floor, if it is moving between floors. Furthermore, the action ‘addRequest()’ should also be assigned to this entity object. The latter should therefore encapsulate the list of floors to visit.

As there are no computation actions, nor guards, which refer to the controlled units ‘door’ and ‘floor Sensor’, we do not specify any entity object to these controlled units.

4. The action ‘determineElevator()’ that selects the most suitable elevator to service the user request, needs accessing to all the elevator entity objects to use the floor sensor, we define the interface object ‘floorSensorInterface’, to handle interaction at every elevator approaching a floor.
the current status of the elevators. It therefore corresponds to a control object, named ‘Scheduler’.

(5) Timing constraints related to door events will be handled by the timer object ‘doorTimer’. This object sends a timeout signals to the use case coordinator object according to the specified timing constraint.

(6) Finally, qualities of service mentioned in the use case text, are mainly related to the optimization of the elevator movement. Usually, to deal with some of these non-functional requirements, we need an object that periodically calculates statistical parameters such as: the average busy time and idle time of each elevator, floor(s) where elevator(s) are very often requested. To this end, we define an application logic object ‘statisticalMeasure’ that encapsulates such computations.

Fig. 6 that depicts the resulting object model shows the necessary objects realizing the use case and theirs links. As previously noted, the coordinator object ‘elevRequestCoordinator’ is an overall decision-making object that determines the overall sequencing for all the objects related to the use case. All these objects are therefore linked to it.

Moreover, the control object ‘Scheduler’ has links to the objects ‘elevStatusPlan’, since it accesses to each elevator entity object in order to use its status information to select the most suitable elevator to the user request. The object ‘statisticalMeasure’ has links to the ‘elevRequestCoordinator’ and ‘elevStatusPlan’ objects.

4. Related works

In recent years, object-oriented techniques have been employed in the development of embedded systems, e.g. RT-UML [5], UML-RT [3], Octopus [21], Artisan Real-Time Perspective [22], etc. There are also several papers in the literature that discuss the process of object identification, e.g. [8,9,11,12,20,23].

In UML-RT (UML for Real-Time), the construction of object and class diagrams is performed by the derivation of scenario diagrams and capsule collaboration from use case scenarios, and the synthesis of object and class diagrams from initial sequence diagrams that only depict interaction between control software and system devices. Software objects are determined using CRC technique [2] or Jacobson’s object categories (interface, control, and entity) [20].

In RT-UML and Artisan Real-Time perspective, objects are identified by the textual analysis of problem statements or use case descriptions.

In Octopus, the system under study is partitioned into subsystems. An analysis phase is performed for each subsystem whose object/class model is built from the problem description by underlying nouns to determine candidate classes. Usually, too many objects/classes are obtained in that manner.

This technique is also applied in [3,9,12] but with the use of automatic tools for natural language analysis of the use case text and the problem statements.

In [23], Rosenberg and Scott extend the approach of Jacobson et al. [20] for identifying objects, using robustness analysis. However, their steps describe, in a very brief form, what needs to be done to put together the main parts of the object diagram. These describing steps do not give any type of detailed instruction on ‘how’.

In [8], an approach to identify objects from use cases is presented. It consists of a four-step rule set. The approach defines the steps to obtain a holistic set of objects. Each use case is transformed in three objects (one interface, one data, and one control). Besides the restricted types of objects, this approach presents some limitations in a sense that it concentrates on what needs to be done rather than addressing ‘how’ it can be done.

In [11], an approach that identifies classes is presented. It is based on goals of use cases without descriptions. The approach produces use case entity diagrams as a vehicle for deriving classes from use cases. However, only domain classes are identified and there is no more global technique that would allow making the transition between the two models in a systematic manner.

Most of these above mentioned tools and methods begin with building classes rather than objects. Our approach focuses on objects and communication between them rather than on classes and associations between classes.

To some extent, our approach complements the existing ones in the sense that in the analysis phase, it begins by the dynamic model instead of the structural one. We believe that is better to build the dynamic model before the static one. If one begins with the dynamic model, it is easier to identify actions, events and also data items, which will be needed in the static model.

5. Conclusion

In this paper we described an approach that bridges the gap between the outside behavioral system description as offered by the use cases and the system structure represented by the object model.

Since most real-time embedded applications have a static structure, it is convenient to see the system under development as a set of communicating objects rather than as a set of classes with associations.

The proposed approach presents a systematic technique for converting use cases into activity diagrams and uses
the latter as a means to identify objects. The semi-formal nature of activity diagrams allows for discovering the necessary objects and their properties (operations and attributes), which are needed for realizing the use case.

During the activity diagram derivation process, detected ambiguities, omissions, impreciseness, and inconsistency in the use case text are removed. This enables enriching the use case model and producing more precise and complete requirements. Moreover, activity diagrams are precise models, easy to understand and work with. They can be used in all the phases of the development process, and hence they constitute a highly reused artifact.

Currently, we are applying our approach in the development of several real-time embedded applications, such as those of the automotive domain and the computer-based manufacturing systems. In addition, we are investigating the extension of the approach to identify not only the objects but also the aspects in an aspect oriented development process.

Acknowledgements

We are very grateful to Professor Zoubir Mammeri from Paul Sabatier University in Toulouse (France), for his contribution to the ideas described in this paper. Furthermore, we would like to thank the anonymous referees whose comments were very useful in improving this paper.

References