

A Process Model for Specifying System Behaviour with UML

Ana Moreira, João Araújo and Fernando Brito e Abreu

Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

2825-114 Caparica, PORTUGAL
TEL: + 351-21-2948536; FAX: + 351-21-2948541
{amm, ja, fba}@di.fct.unl.pt

Abstract

Software development projects grew a reputation for poor quality. This situation is in part due to the lack of appropriate mechanisms to identify and express functional requirements in a flexible yet rigorous fashion. UML is a standard modelling language that is able to specify applications in different levels of abstractions as it provides a wide range of notations. Among them, we have collaborations that serve to realise use cases, a powerful abstraction concept. The behaviour part of a collaboration is rendered using sequence or collaboration diagrams. However, the lack of abstraction and refinement mechanisms compromises the understandability and modularity of a specification. In general, we can say that abstraction and refinement mechanisms help obtaining a more maintainable system. Our aim is to provide abstraction and refinement mechanisms accomplished by proposing a modelling process.

Keywords: UML, sequence diagrams, collaborations

Introduction

It is often the case that software projects fail to meet user expectations. When looking for root causes of this sorrow state we often come across the problem of inadequate requirements elicitation and evolution. Thus, it is of utmost importance the availability of powerful formalisms to help expressing functional requirements iteratively and incrementally.

The Unified Modelling Language (UML) provides several concepts, techniques and respective notations to be used at different levels of abstraction throughout the development process [2]. For example, at a higher abstract level use cases, as proposed by Jacobson, are used to describe the outwardly visible requirements of a system [8]; they describe functional requirements of a system, are used in the requirements analysis phase of a

project and contribute to test plans and user guides [13]. Use cases are a fundamental tool to help identify a complete set of user requirements. A use case describes a complete transaction, including error situations and exceptions, and normally involves several objects and messages. Software developers are easily seduced by the simplicity and potential of use cases; they claim that use cases are an easily understood technique for capturing requirements.

Use cases are “a society of classes, interfaces and other elements that work together to provide some cooperative behaviour” [2]. They can be refined through collaborations, each consisting of two parts: the structural and the behavioural ones. The structural part is specified in a class diagram. The behavioural part is rendered using one or more interaction diagrams, like, for example, sequence diagrams. A sequence diagram shows how messages exchanged among considered objects are ordered in time.

In general, abstraction and refinement mechanisms help obtaining more maintainable systems. Sequence diagrams are limited since they only provide one level of abstraction. Besides, there is no explicit support, in UML, to their refinement. The inability to represent refinement in those diagrams compromises the understandability and modularity of a specification, and as such, the overall quality of this kind of behaviour models.

The aim of this paper is to investigate the specification steps from uses cases to sequence diagrams, proposing a process to model our findings. This process should include a mechanism to support refinement of sequence diagrams. The process should be iterative and incremental. The availability of a complete set of detailed use cases and collaborations is not mandatory before we start drawing sequence diagrams and their refinements. We can start with the subset of the best understood and more representative informal requirements, define its use cases and respective collaborations, and specify the initial

sequence diagrams for each collaboration, and then start their refinement.

In Section 2 we describe the process model. In Section 3 we apply the process to a case study. In Section 4 we discuss some related work. Finally, in Section 5, we draw some conclusions.

The process

Our goal is to investigate how sequence diagrams can be refined so that we can specify, step by step, the behaviour of a system using UML. Our idea is to propose

a process that derives object-oriented specifications for the behaviour part of collaborations, starting from a use case model. Each collaboration is translated into a set of sequence diagrams, each one offering a partial view of the objects involved in that transaction. The full integration of all these views gives the complete functionality of the system.

The process, depicted in Figure 1, is composed of three main tasks: define the use case model, specify the collaborations and specify the formal model or build a prototype. In the next sections we describe each task in terms of its subtasks.

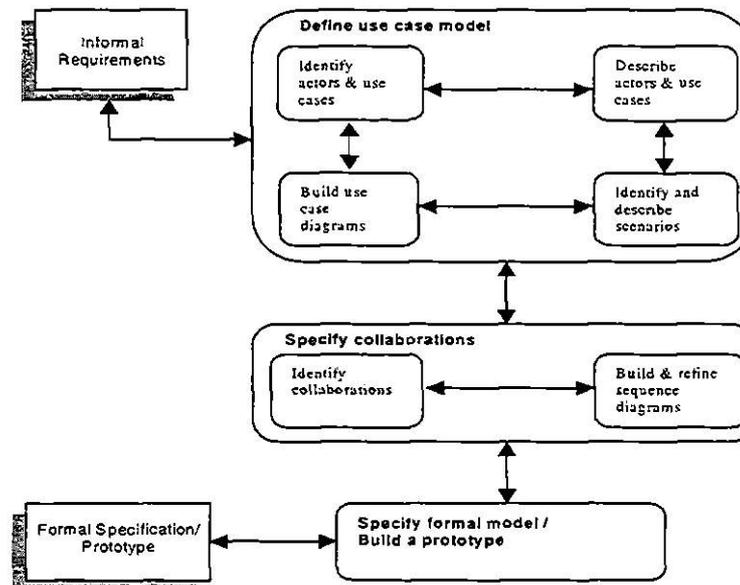


Figure 1. The process model

The process is iterative and incremental. We do not propose that a complete set of use cases and collaborations be found and described before we start drawing sequence diagrams and their refinements. Instead, we can start with the subset of the informal requirements we understand better, define its use cases and respective collaborations, specify the initial sequence diagrams for primary scenarios and from here refine them as shown in Section 4. In later iterations we deal with the secondary scenarios.

Task 1: Define the use case model

To define the use case model we need to start by identifying the actors and corresponding use cases of the system. An actor represents a coherent set of roles that users of the use cases play when interacting with the use cases [2]. A use case is a description of a set of sequences of actions that a system performs that yields an observable

result of value to an actor. The major subtasks of this task are: identify actors and use cases; describe actors and use cases; identify and describe scenarios; build the use case diagram.

Subtask 1.1: Identify actors and use cases. The first subtask is to identify the actors of the system. Actors are anything that interfaces with our system. Some examples are people, other software, hardware devices, data stores or networks. Each actor assumes a role in a given use case. Then we need to go through all the actors and identify use cases for each one. Use cases describe things actors want the system to do. In the UML a use case is always started by an actor, however there are situations where we have found it useful to initiate use cases from inside the system. These situations usually appear when the functionality is time related.

Subtask 1.2: Describe actors and use cases. Having identified all the actors and use cases, we have to describe them. Each use case must include details about what has to be done to accomplish the functionality of the use case. These include the basic functionality, but also alternatives, error conditions, preconditions and post conditions. The use case may include conditionals, branching and loops.

In the first iteration we can start by giving a name and a brief description, one or two sentences long, to each actor and use case. In later iterations we can describe each use case using natural language, scenarios, or pseudo-code. (The Rational Unified Process gives a basic format for a use case [9].) We prefer to use scenarios, described as a list of numbered steps. A scenario is a particular path of execution through a use case.

Subtask 1.3: Identify and describe scenarios. Use cases can be fully described using a primary scenario and several secondary scenarios, depending on the use case complexity. The primary scenario represents the main path of the use case, i.e. the optimistic view. If everything goes well, then what happens is the primary scenario. The secondary scenarios describe alternative path, including error conditions and exception handling. Therefore, one important step here is to identify and describe, for each use case, its primary and secondary scenarios. The initial iterations should handle only primary scenarios, leaving secondary scenarios for later iterations.

Subtask 1.4: Build the use case diagram. The use case model shows the system boundaries, the actors and the use cases. Actors may be related between themselves and with the use cases they activate. Some use cases can also be related to other use cases.

A use case diagram uses four types of relationships: generalization, include, extend and association. While generalization is a relationship that can be used between actors and between use cases, include and extend are relationships between use cases. On the other hand, an association is the communication path between an actor and a use case. Actors that have similar roles, and therefore activate (are associated with) the same subset of use cases, can inherit from each other, so that the complexity (number of associations from actors to use cases) of the diagram can be alleviated. Include allows the insertion of additional behaviour into a base use case that explicitly describes the insertion. Extend allows the insertion of additional behaviour into a base use case that does not know about it [12].

Task 2: Specify the collaborations

The second task is composed of the two main subtasks: identify collaborations; build and refine sequence diagrams.

Subtask 2.1: Identify collaborations. Having specified the use cases we can start identifying and associating collaborations to realise them. The collaborations will realise the use cases through class diagrams and interaction diagrams (i.e. sequence and collaboration diagrams). We associate a collaboration to each use case and start modelling it using a sequence diagram. The new objects identified in each sequence diagram will originate a corresponding class in the class diagram. Therefore, the class diagram can be built in parallel with the sequence diagrams. The collaboration diagrams can then be generated from each sequence diagram by using any CASE tool such as the Rational Rose 2000 [11].

Subtask 2.2: Build and refine sequence diagrams. Each sequence diagram draws a scenario. In the first iterations we only deal with the primary scenarios, leaving the secondary scenarios for later iterations, when the main functionality of the system is already specified.

The different levels of abstraction of sequence diagrams depend on the kind of objects that we want to have at each level. We propose that the most abstract level contains only one object that represents the system. The next level contains boundary objects, the next one contains control objects and, the final one, provides the entity objects.

Having this in mind, we can follow the steps below to refine each sequence diagram of a collaboration:

- Consider the system as a black box, represented in the diagram as an object, and identify the interactions between it and the users (actors) that activate the scenario;
- Look at the object that represents the whole system and "open" it to show a boundary object and again an object that represents the rest of the system (constituted of control and entity objects);
- Draw another sequence diagram where we show the boundary and the control objects, and the object that represents all the entity objects;
- Finally, another sequence diagram has to be drawn to show the entity objects.

Each of the sequence diagrams above can have levels of abstraction in terms of the "granularity" of messages, i.e., a message can be refined into a subsequence of messages between two objects.

Other refinements include, for example, the refinement of the boundary object into its component objects, if any. Entity objects can also be complex objects that we may want to decompose in later iterations.

Task 3: Specify the formal model/Build the prototype

At this point we can follow different directions, depending on the organization interests and the application being built. One alternative is to keep specifying the system building a formal, or at least rigorous, model. We have been working on that line of research, by formalizing the UML models using LOTOS [3], Object-Z [6], SDL [7]. The formalisation process is not always straightforward and depends on the skills and familiarisation with the formal description techniques of the analysts involved in the specification. Therefore, derivation rules should be provided to generate a corresponding formal specification of a collaboration, in order to encourage and speed the formalisation process.

Another different perspective is to build a prototype of the future system by using an evolutionary approach. The main advantages are to accelerate the delivery of the system and stimulate the user engagement with the system. Here we can use high-level languages for prototyping as for example Smalltalk, Lisp, Prolog and 4GL.

Applying the process to a case study

To exemplify the process described in the previous section consider the case study we have chosen [4].

"In a road traffic pricing system, drivers of authorised vehicles are charged at toll gates automatically. They are placed at special lanes called green lanes. For that, a driver has to install a device (a gizmo) in his vehicle. The registration of authorised vehicles includes the owner's personal data and account number (from where debits are done automatically every month), and vehicle details. A gizmo has an identifier that is read by sensors installed at the toll gates. The information read by the sensor will be stored by the system and used to debit the respective account. The amount to be debited depends on the kind of the vehicle.

When an authorised vehicle passes through a green lane, a green light is turned on, and the amount being debited is displayed. If an unauthorised vehicle passes through it, a yellow light is turned on and a camera takes a photo of the plate (that will be used to fine the owner of the vehicle).

There are green lanes where the same type vehicles pay a fixed amount (e.g. at a toll bridge), and ones where the amount depends on the type of the vehicle and the distance travelled (e.g. on a motorway). For this, the system must store the entrance toll gate and the exit toll gate."

Define the use case model

A use case model shows a set of actors and use cases and the relationships among them; it addresses the static use case view of a system. In our example, the actors identified are:

- Vehicle-driver: this comprises the vehicle, the gizmo installed on it and its owner;
- Bank: this represents the entity that holds the vehicle owner's account;
- Operator: this may change the values of the system, and ask for monthly debits.

The use cases identified are:

- Register a vehicle: this is responsible for registering a vehicle and communicate with the bank to guarantee a good account;
- Pass a single tollgate: this is responsible for reading the vehicle gizmo, checking on whether it is a good one. If the gizmo is ok the light is turned green, and the amount to be paid is calculated and displayed; if the gizmo is not ok, the light turns yellow and a photo is taken.
- Pass a two-point tollgate: this can be divided into two parts. The in toll checks the gizmo, turns on the light and registers a passage. The out toll also checks the gizmo and if the vehicle has an entrance in the system, turns on the light accordingly, calculates the amount to be paid (as a function of the distance travelled), displays it and records this passage. (If the gizmo is not ok, or if the vehicle did not enter in a green lane, the behaviour is as in the previous case.)
- Pay bill: this, for each vehicle, sums up all passages and issues a debit to be sent to the bank and a copy to the vehicle owner.

Having identified and briefly described use cases, we need to identify their primary scenarios. Each use case is composed of a primary scenario, obviously, and several secondary scenarios. For example the use case "pass a single toll gate" has the primary scenario "pass single toll gate ok" and the secondary scenarios "pass single toll gate without a gizmo" and "pass single toll with an invalid gizmo". In this paper we will use the primary scenario to illustrate the process. Figure 2 shows the primary scenario "pass single toll gate ok".

- | | |
|----|---|
| 1. | The use case starts when the vehicle-driver passes a single toll. |
| 2. | The single tollgate reads the gizmo identifier. |
| 3. | The system verifies the gizmo identifier. |
| 4. | The system turns the light green, calculates the amount to be paid and displays it. |
| 5. | The system stores the usage details and the use case ends. |

Figure 2. "Pass single tollgate ok" primary scenario

Secondary scenarios are described in a similar way. The set of all use cases can be represented in a use case diagram, where we can see the existing relationships between use cases and the ones between use cases and actors. Figure 3 shows the use case diagram of the road traffic system.

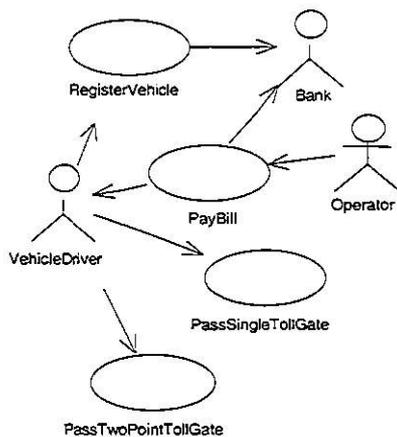


Figure 3. The use case diagram of the Road Traffic Pricing System

Later versions of the use case diagram could show relationships between use cases, in particular some of the use cases share a common set of events in the beginning (which could be shown by adding an extra use case related to the original use cases with the "include" relationship). Extend relationship could also be applied to deal with error situations, for example.

Specify collaborations

Collaborations realise use cases, through a realisation relationship (represented by a dashed arrow). To exemplify this, we choose the use case `PassSingleTollGate`, which deals with the three scenarios already mentioned in the previous section. The

primary scenario deals with authorised vehicles and the two secondary scenarios deal with non-authorised vehicles. The associated collaboration for that is `PassSingleTollGateManagement`. Figure 4 shows the realisation of the use case by that collaboration.

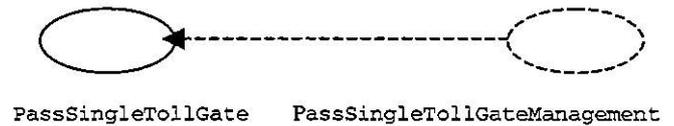


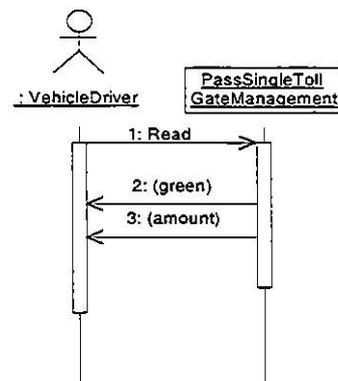
Figure 4. The realisation of the use case for vehicle passing a single tollgate

In the next Section, we show in detail the process concerning the refinement of sequence diagrams.

Build and refine sequence diagrams. In a sequence diagram, objects are shown as icons whose naming scheme takes the form `objectName:ClassName`. However, the name of the class can be omitted, as in Figure 5. Arrows represent the messages. Messages are numbered and may carry arguments, between brackets.

Figure 5 shows the initial sequence diagram for the primary scenario authorised vehicles, passing a single toll (`PassSingleTollGateOk`), with the actor `:VehicleDriver` and the object that represents the collaboration. The system reads the gizmo and, if this is OK, the actor `VehicleDriver` sees the light green and the amount to be paid in the display. This represents the externally visible behaviour of the system.

Figure 5. Initial sequence diagram



As a rule of thumb, boundary objects receive all the events to the system. The system outputs are also made available through this type of objects. In the first iteration, and to start with, we can only represent the interaction point, without having to detail the exact boundary objects.

Figure 6 shows the sequence diagram with the boundary object SingleToll.

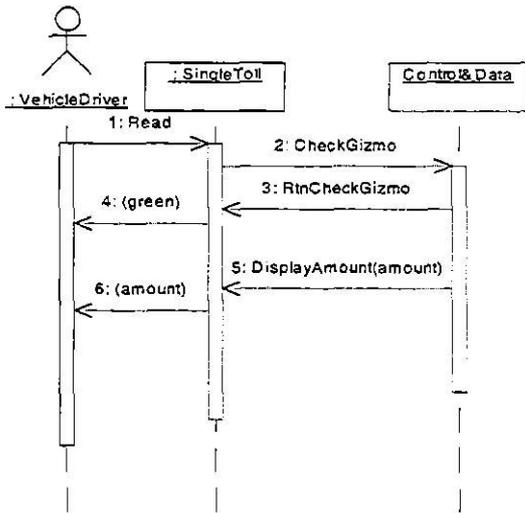


Figure 6. Sequence diagram with boundary object

Boundary objects should only be responsible for accepting inputs to the system or displaying outputs from the system. Therefore, we need a control object whenever we have complex functionality to deal with. Note that control objects are not always needed. As a rule, we may just ignore them to start with, and then add them if the boundary objects handle the major decisions of the collaboration. Other strategy is to accept a control object no matter the complexity of the functionality of the collaboration, and in a “validation step” remove the ones that we see as unnecessary, removing all those that only play the role of intermediary, i.e., those that receive an event and delegate that same event, without processing it. We will follow this strategy. Figure 7 shows the sequence diagram with a boundary and a control object.

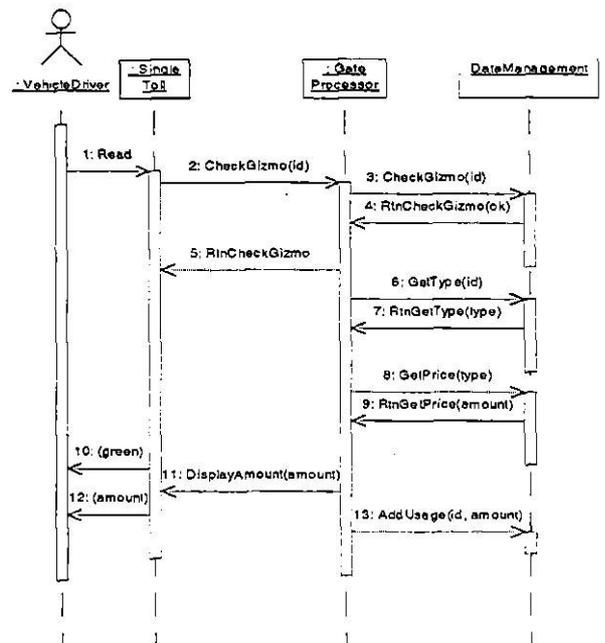


Figure 7. Sequence diagram with boundary and control objects

Now we need to deal with the DataManagement object. Having dealt with the two external layers (boundary and control) we have to identify the entity objects, i.e., the key abstractions of the system. Entity objects hold the information that must be provided for the completion of the functionality of the scenario. Figure 8 shows the sequence diagram with the entity objects.

From here we could jump to task 3 (specify the formal model or build the prototype). In later iterations we could add still more detail to the sequence diagram. In particular, there is more we can do about boundary objects. We know that a toll gate has to have a sensor to detect the vehicles and to read their id number. We can also see that a light may be turned green or yellow, depending on whether we are authorised users or unauthorised ones. Also, we see the amount to be paid being shown on a display. Finally, if we want to deal with unauthorised vehicles, a camera should photograph their plate numbers. Therefore, single toll gate is composed of: sensor, light, display, camera. Figure 9 refines the previous sequence diagram by incorporating these objects. As we are dealing with the primary scenario we do not need to represent the camera object in this diagram.

In summary, after the sensor reads the gizmo, this must be checked to see if it is valid; the toll gate turns the light green and shows the amount that will be debited from the

vehicle's owner bank account. The amount must be calculated according to the type of the vehicle and displayed. Finally, the passage must be recorded in usage details.

Specify the formal model/Build a prototype

The objective of this work is not to describe the formalisation process or to build the prototype. In the former alternative, the approach described in [1], which formalises collaborations using Object-Z, can be applied here. An evolutionary prototype can be built by using adequate tools, e.g. 4GL.

Related work

There is some work that describes a process to specify system behaviour. Dano *et al.* [5] present an approach based on the concept of use case to support the requirements engineering process. This is a "domain expert-oriented" where the domain expert actively participates in the specification of the use cases. These are described by tables and Petri nets. Rolland and Archour [10] have developed CREWS. This is a model of use case together with a guidance process for authoring use cases. The approach involves contextual description of the use cases and writing and refining scenarios. Sendall and Strohmeir [14] describe an approach that supplements use case descriptions with operation schemas. These are declarative specifications of system operations written in OCL [15].

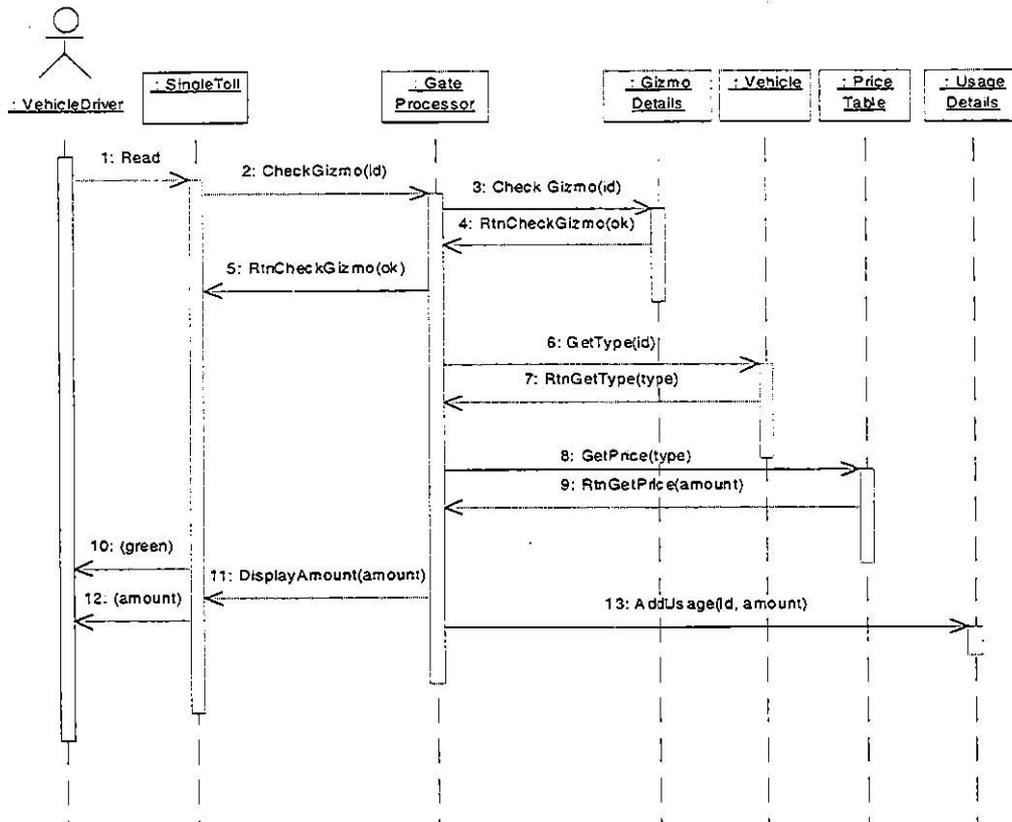


Figure 8. Sequence diagram with boundary, control and entity objects

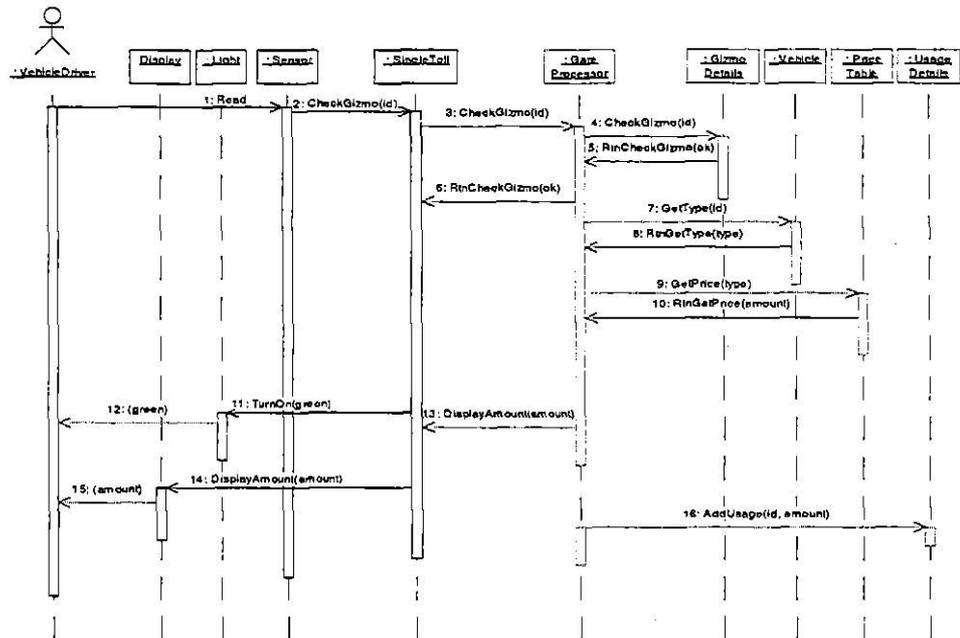


Figure 9. Refined sequence diagram showing the Single Toll components

Our previous work [1] shows the formalisation of use cases and respective collaboration using Object-Z, but refinement is not considered.

Related areas of interest are the transformation of dynamic models and the construction of supporting tools. In [17] he proposes a mechanism to transform sequence diagrams into state charts. In [16] work has been done where the Maude system (based on rewriting logic) is used to automate transformations of UML behaviour models, and can be applied to our process.

Conclusions and future work

The process described in this paper provides a systematic way to specify the behaviour of a system, starting from use cases, identifying collaborations, and describing the respective sequence diagrams. These are refined to different levels of abstraction according to the kind of objects represented in each level. This improves the work of the analyst as he/she can look at the system from different levels of abstraction, enhancing the communication among the different members of the development team. The outcome is

a quality result is a better quality for the specification.

For future work we are planning to formalise and automate the refinement process. The formalisation is important if we want to guarantee consistency the different levels of abstraction of the diagrams. The automation is essential as updating the models is a highly time-consuming and error-prone activity if done manually. Other related area of interest is the transformation of dynamic models. We are investigating, for example, how sequence diagrams can be transformed into state diagrams.

References

- [1] Araújo, J. and Moreira, A.: "Specifying the Behaviour of UML collaborations Using Object-Z", America Conference on Information Systems, Long Beach, California, August 2000.
- [2] Booch, G., Rumbaugh, J. and Jacobson, I.: The Unified Modeling Language User Guide, Addison-Wesley, Reading, Massachusetts, 1998.
- [3] Brinksmas, E.: LOTOS: a Formal Description Technique Based on Temporal Observational Behaviour, ISO 8807, 1988.

- [4] Clark, R. and Moreira, A.: Constructing Formal Specifications from Informal Requirements, *Software Technology and Engineering Practice*, IEEE Computer Society, Los Alamitos, California, July 1997, pp. 68-75.
- [5] Dano, B., Briand, H. and Barbier, F.: "An Approach Based on the Concept of Use Case to Produce Dynamic Object-Oriented Specifications", *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, 1997.
- [6] Duke, D., King, P., Rose, G. A. and Smith, G.: "The Object-Z Specification Language," Technical Report 91-1, Department of Computing Science, University of Queensland, Australia, 1991.
- [7] Ellsberger, J., Hogrefe, D. and Sarma, A.: *SDL*, Prentice-Hall, 1997.
- [8] Jacobson, I.: *Object-Oriented Software Engineering -- a Use Case Driven Approach*. Addison-Wesley, Reading Massachusetts, 1992.
- [9] Jacobson, I., Booch, G. and Rumbaugh, J.: *The Unified Software Development Process*, Addison-Wesley, 1999.
- [10] Rolland C. and Achour, B.: "Guiding the Construction of Textual Use Case Specifications". *Data and Knowledge Engineering Journal*, Vol. 25, N° 1-2, North-Holland, March 1998.
- [11] ROSE, CASE tool, <http://www.rational.com/products/rose>.
- [12] Rumbaugh, J., Jacobson, I. And Booch, G.: *The Unified Modeling Language Reference Manual*, Addison-wesley, 1999.
- [13] Schneider, G. and Winters, J. P.: *Applying Use Cases – A Practical Guide*. Addison-Wesley, 1998.
- [14] Sendall, S. and Strohmeier, A.: "From Use Cases to System Operation Specifications". *UML 2000 – Advancing the Standard, Lecture Notes in Computer Science*, Vol 1939, Springer-Verlag, October 2000.
- [15] Warmer, J. and Kleppe, A.: *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998.
- [16] Whittle, J., Araújo, J., Alemán, J.L.F., and Toval, A.: *Rigorously Automating Transformations of UML Behaviour Models*, *Workshop on Dynamic Behaviour, UML 2000*, York, October 2000.
- [17] Whittle, J. and Schumann, J.: *Generating Statecharts from Scenarios*, *Proceedings of the International Conference on Software Engineering*, Limerick, Ireland, 2000.