GENERATING OBJECT-Z SPECIFICATIONS FROM USE CASES

Ana Moreira and João Araújo
Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
2825 Monte da Caparica
PORTUGAL
TEL: + 351-1-2948536; FAX: + 351-1-2948541
{amm | ja}@di.fct.unl.pt

Keywords: Object-oriented system analysis, modelling formalisms and languages, UML

Abstract: The importance of use cases has been growing for the last few years. We believe they are important to help developers capturing requirements. The work presented here formalises use cases using basic temporal logic to define history invariants within Object-Z class schemas. This is accomplished by proposing a set of formal frameworks integrated within a process.

1. INTRODUCTION

Use cases, as proposed by (Jacobson 1992), describe functional requirements of a system, helping to identify the complete set of user requirements. A use case is a generic transaction, normally involving several objects and messages. Industrial software developers are easily seduced by the simplicity and potentiality of use cases; they claim that use cases are an interesting and easily understood technique for capturing requirements.

Use cases reduce the complexity of the requirements-capture process, as they give us a systematic approach to fully identify the requirements of a system. The classification of users into actors, on one hand, and the description of what each actor expects from and gives to the system, on the other hand, is a major help during the elicitation process. However, this is not enough to guarantee that the requirements do not contain errors, ambiguities, omissions and inconsistencies. These drawbacks can only be identified and corrected early in the development process if formal description techniques are used.

The goal of this paper is to specify use cases formally, using the object-oriented paradigm. The starting point is a subset of UML (Booch 1998). We will adopt sequence diagrams to represent instances of a use case. Sequence diagrams show the time ordering of messages exchanged between the objects involved in a use case.

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, and automated, to generate a corresponding formal framework of a use case, in order to encourage and accelerate the formalisation process. These rules can be given using any formal specification language. Here, we have chosen Object-Z (Duke 1991).

2. RELATED WORK

Several methods combine formal specification languages with an object-oriented method (Telelogic 1998, Sinclair 1996, Reed 1996, Kuusela 1993). However, while we use Object-Z to formalise use cases during the requirements capture they use SDL, but only for the design phase. For the analysis phase they use OMT (Rumbaugh 1991). Examples are the SOMT method (Telelogic 1998) and the SISU method (Braek 1996).

The ROOA method (Moreira 1996) proposes an integrated approach to build a formal object-oriented specification from informal requirements. This is accomplished by first creating a user-centred model and from there creating a system-centred model. The user-centred model is a set of user views, each one showing the interactions between the users and the system. The interaction within the system is only represented in the system-centred model. We model the use cases fully, not only the externally visible interactions.

Our work has some similarities with viewpoints (Kotonya 1996, Nuseibeh 1994). However, each viewpoint is defined by different participants using different notations. The end result is, therefore, difficult to integrate. We use the same notation to formalise all use cases.

3. OBJECT-Z

Object-Z is a well-known extension of Z (Hayes 1987, Spivey 1992) to incorporate object-oriented concepts (Meyer 1988). It is well documented and it serves for the purposes of this paper. Object-Z has been used in many real applications, including real-time systems in the telecommunications area. It is a model-based language that has its roots, like Z, in set theory; its most important feature is the *class schema*. A class schema takes the form of a named box, optionally with generic parameters. It extends the graphical component of Z (boxes) to define its classes, providing an immediate visual indication of the scope of the definition. Figure 1 shows the form of the Object-Z "box".

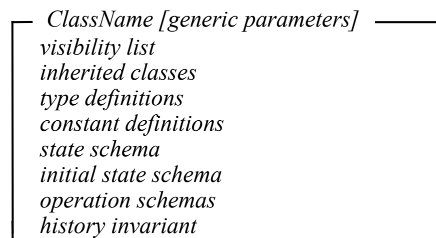

Figure 1: Object-Z class schema

The components of this box are:

- a list of visibility that restricts access to attributes and operations;
- a list of inherited classes;
- a list of type and constant definitions;
- a state schema which defines the class invariant and its state attributes;
- an initial state schema that specifies the initial state of the objects of the class;
- a set of operation schemas that specifies the pre and post conditions of the operations of the class;
- a history invariant that constrains the order of the operations and is defined using temporal logic.

The class schema extends Z's typing scheme by permitting classes to act as types.

4. OVERVIEW OF THE PROCESS

In this paper we propose a process that derives a formal object-oriented specification, using Object-Z, from a set of informal requirements. This process is shown in Figure 2.

At a higher level of abstraction we propose two main tasks: define a use case model and specify a formal model. The first is composed of three subtasks: identify a list of use cases, describe use cases and build sequence diagrams for each use case. The second task is

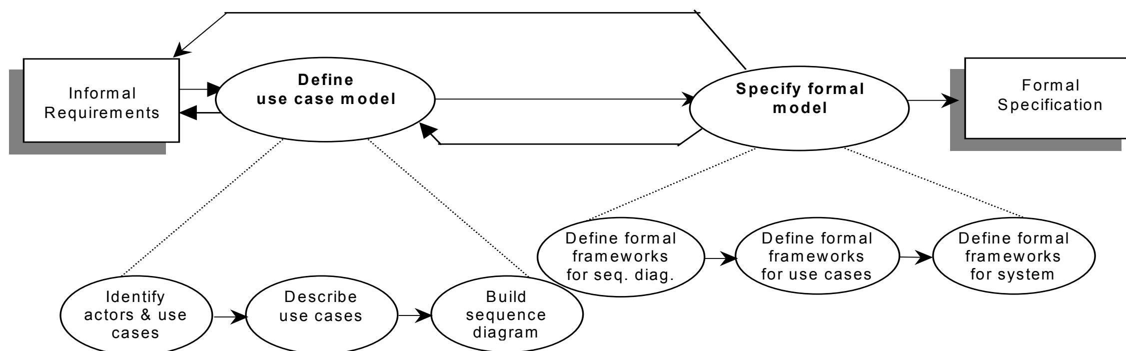

Figure 2: Core of the use case formal process

also composed of three main subtasks: define formal frameworks for sequence diagrams, compose them to formalise use cases and, finally, compose the end result to obtain a full specification. A formal framework is extracted using a pre-defined set of rules that is part of the process. These rules are defined using temporal logic and are applied to each sequence diagram. The formal framework is a template to be used as a basis to produce then a full class schema specification. To have an abstract version of a class schema has the advantage of increasing understandability, thus improving the formalisation process.

The process is iterative and incremental. We do not propose that a complete set of use cases be found and described before we start drawing sequence diagrams and specifying Object-Z class schemas. Instead, we can start with the subset of the informal requirements we understand better, define its use cases, translate these into sequence diagrams and from here generate Object-Z class schemas. Each use case, translated into sequence diagrams, offers partial views of several objects. These views will be integrated to show the complete functionality of the system. As we understand more of the requirements, we can introduce either more detailed information in a use case, or add new use cases to our system. This new information can either be added to existing sequence diagrams or new ones can be created and all the changes will be propagated into the Object-Z class schemas.

5. APPLYING THE PROCESS

5.1 The case study

The case study we have chosen is taken from (Clark 1997).

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.

5.2 Define the use case model

Our goal is to identify the use case model of the system, and for each one, to formalise the associated sequence diagrams.

To identify the use case model we need to start by identifying the actors and corresponding use cases of the system. According to (Booch 1998), an actor represents a coherent set of roles that users of the use cases play when interacting with the use cases. 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. 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. Figure 3 shows the use case diagram of the road traffic system.

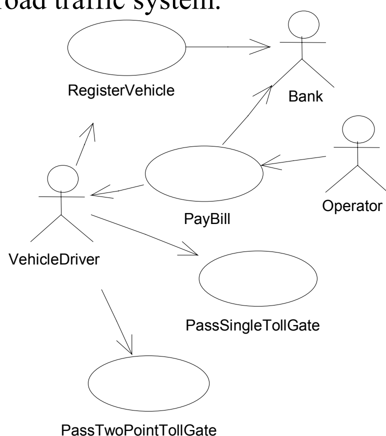

Figure 3: The use case diagram of the Road Traffic Pricing System

The actors identified are:

- Vehicle Driver: this comprehends 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 toll gate: 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 toll gate: 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.

Next step is to draw the sequence diagrams for each use case. We choose the use case *PassSingleTollGate*, which deals with two situations: authorised vehicles and non-authorised vehicles. Given that UML allows alternatives, a single sequence diagram can be used to handle both situations. For simplicity, as Figure 4 shows, we will deal only with an authorised vehicle passing a single toll (*PassSingleTollGateOk*).

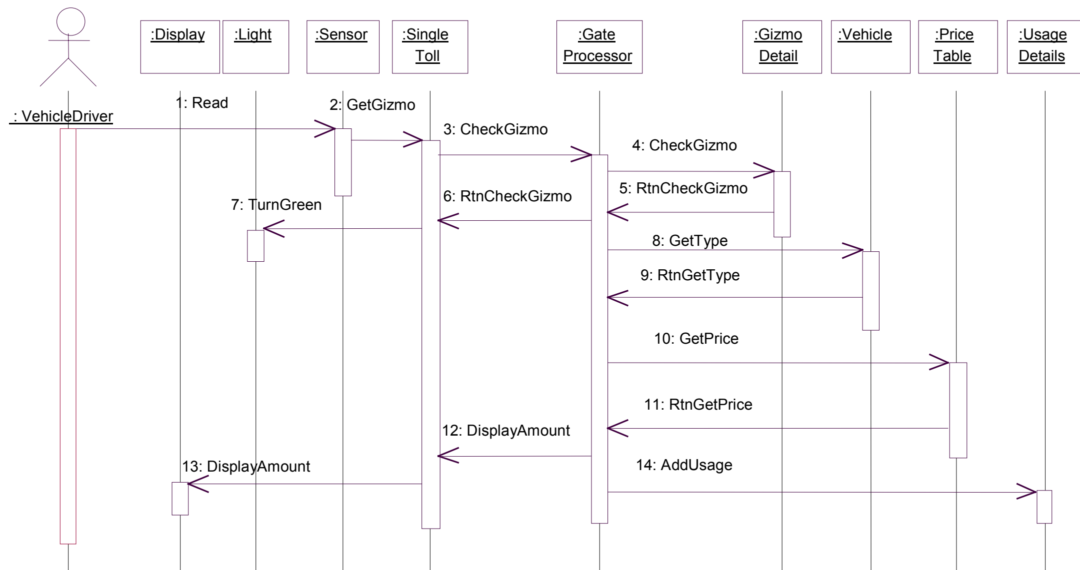

Figure 4. Sequence diagram depicting an authorised vehicle passing a single toll gate

As we build the sequence diagrams, objects, services and message passing are identified.

5.3 Specify the formal model

In this task, we build a formal object-oriented specification centred on the use cases. In a later stage, which is out of the scope of this paper, we will be interested in a specification centred on the objects (Araújo 1996).

Our specification, centred on use cases, is an initial formal specification that helps us reasoning about the final specification earlier. Without this, the formalisation of the use would only be possible after specifying all the classes identified, delaying unnecessarily the formalisation process. Therefore, we do not have to formalise all the classes beforehand to be able to formalise the use cases.

The rules defined to generate an Object-Z framework from the associated sequence diagrams of a use case are based on safety, guarantee and response properties of programs that can be specified by temporal logic formulas (Manna 1992). The temporal logic operators used are $\Box$ (always) and $\Diamond$ (eventually).

We can then define:

1. *Safety Property*: can be specified by a safety formula. A safety formula is any formula that is equivalent to a canonical safety formula $\Box p$ (p always holds). Usually, safety formulas represent invariance of some state property over all the computations.
2. *Guarantee Properties*: can be specified by a guarantee formula. A guarantee formula is equivalent to a canonical formula of the type $\Diamond p$. This states that p eventually happens at least once.
3. *Response Properties*: can be specified by a response formula. A response formula is equivalent to a canonical formula of the type $\Box\Diamond p$. This states that every stimulus has a response. An alternative formula is $\Box(p \rightarrow \Diamond q)$, which states that every p is followed by a q, that is, q is a guaranteed response to p.

These properties can be classified into safety and progress (or liveness). A safety

property states that a requirement must always be satisfied in a computation. Progress properties can be either guarantee or response. The progress properties specify a requirement that should eventually be fulfilled. Therefore, they are associated with progress towards the fulfilment of the requirement.

A history invariant can specify progress issues by showing how the various messages interact, for example, when specifying the priority, or the order in which messages may or may not happen. Sequence diagrams show the message passing and synchronisation among objects, which can naturally be expressed by temporal logic. Therefore, it is practicable then to translate process sequences into history invariants.

A sequence diagram can be formulated as a class schema. This contains instances of the participant classes of the sequence diagram, and defines a history invariant that represents the sequence of messages itself. The sequence diagram *PassSingleTollGateOK* is used to illustrate the mapping rules described below.

1. A sequence diagram can be mapped into Object-Z as a class schema where its label is derived from the sequence diagram name defined in the respective template. In the example, the class schema name generated is *PassSingleTollGateOK*.
2. The objects that participate in the sequence diagram are specified in the state schema definition part. Anonymous objects must be given a name at this point, which will be the state variables. Objects without classes (or types) will be declared with type *UndefinedClass*.
3. All the objects have to be initialised. Therefore, the initial state schema of the class consists of a conjunction of application of *Init* messages to the objects that participate in the sequence diagram.
4. The message passing of the sequence diagram and its ordering is converted into a history invariant that is expressed by a temporal logic formula.
5. Each message, in a sequence diagram, is passed from a sender object to a receiver object, can have an associated condition and has an order number. Object-Z uses the pre-defined operator **op** to specify messages. If we define α_i as a message being sent from a sender to a receiver, we can formalise it as ($\mathbf{op} = o_{i+1}.m_i$) or ($condition_i \wedge \mathbf{op} = o_{i+1}.m_i$) where $1 \leq i \leq n$. Then we can define the rules below. In the case of a sequential message passing, we have:
   - if there is only one message, this can be mapped to the canonical formula $\Box\Diamond\alpha_i$, where i = 1; otherwise,
   - if there is a sequence of messages, the general response form is $\Box(\alpha_i \rightarrow \Diamond\beta)$, where α_i represents the first message and β the rest of the sequence. β has two forms:
     - α_j with $1 < j \leq n$, to deal with the last message, and
     - $\alpha_j \rightarrow \Diamond (\alpha_{j+1} \rightarrow \ldots \Diamond (\alpha_{n-1} \rightarrow \Diamond \alpha_n)\ldots)$ where $1<j \leq n$.

   In the case where we have concurrency:
   - if there is only one group of concurrently messages, this can be mapped to the canonical formula $\Box\Diamond\gamma_k$, where k = 1. Where $\gamma_k = \bigwedge^{n}_{p=1} \alpha_p$. Otherwise,
   - if we have two or more groups of concurrent messages being sent (that is, messages with the same order number), the general form is $\Box(\gamma_k \rightarrow \Diamond\beta)$, where γ_k is as before and β is:
     - γ_j with $1 < j \leq n$, to deal with the last group of messages.
     - $\gamma_j \rightarrow \Diamond (\gamma_{j+1} \rightarrow \ldots \Diamond (\gamma_{n-1} \rightarrow \Diamond \gamma_n)\ldots)$ where $1<j \leq n$.

The Object-Z framework of the sequence diagram *PassSingleTollGateOK,* illustrated in Figure 5, is obtained by applying the rules above.

Having specified and formalised sequence diagrams for each use case identified, we are able to formalise the use case itself. Each use case can be mapped into a class schema framework following the rules below:

1. The name of the class schema has the same name o the use case. For example, in

our case the class schema name is *PassSingleTollGate.*

2. Each use case class schema inherits all the class schemas derived from the respective sequence diagrams.
3. Explicit renaming of variables is the responsibility of the specifier, if this is necessary.

In the example, the use case *PassSingleTollGate* generates the class schema framework shown in Figure 6, by applying the rules above.

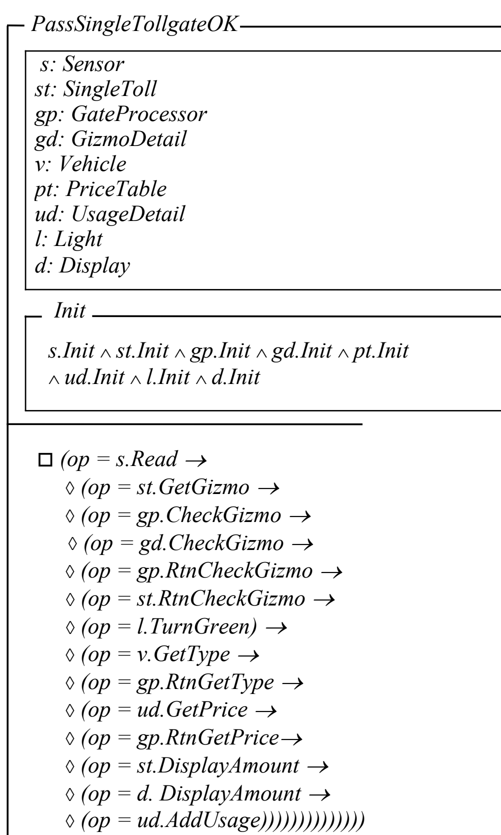

Figure 5. Class schema framework of *PassSingleTollGateOK*

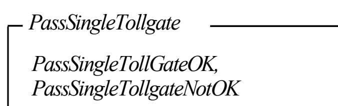

Figure 6. Class schema framework of the use case *PassSingleTollGate*

Finally, the use case diagram can also be mapped into a class schema framework using similar rules:

1. The name of the class schema framework has the same name o the use case diagram. In the example, the name of the class schema will be *RoadPricingSystem*.
2. This class schema inherits all the class schema frameworks derived from the respective use cases.
3. Use cases can be organised mainly by specifying generalisation, include and extend relationships. Generalisation is directly supported by Object-Z. The relationships extends and includes are both formalised as two sequence diagrams communicating with each other. The sequence diagram corresponding to the base use case calls the sequence diagram corresponding to either the extension (for the extend relationship) or supplier (for the include relationship).
4. Explicit renaming of variables is the responsibility of the specifier, if this is necessary.

Figure 7 shows the class schema of the use case diagram of the road traffic pricing system.

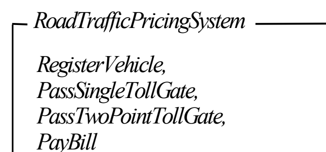

Figure 7. Class schema of the road traffic pricing system

With the classes, messages and services identified we can build a class diagram and complete it with class attributes. This information, together with our use case centred specification, can be used to build a formal specification centred on the objects that compose the system.

6. CONCLUSION

The process described in this paper provides a set of rules to transform use cases (and sequence diagrams) into an object-oriented formal notation (Object-Z). The work described here has the exclusivity to introduce rules to transform the use cases into Object-Z's class schemas. The integration of the two approaches is synergetic because the sum of the advantages of these approaches is greater than if they are considered in isolation. Some of the advantages identified are:

- This encourages the formalisation of the system at early stages;
- This normalises different notations into one precise mathematical notation;
- This favours traceability;
- This promotes a deep reasoning about the system, as the language has a mathematical semantics.

Nevertheless, more work must be done. In particular, we need extend our formal frameworks to handle concurrency from the point of view of an object that receives simultaneously the same message from different senders. Also, we need to integrate this work with a formalisation process that builds a specification centred on objects. To improve the process, reverse engineering should be defined, i.e., from class schemas we should obtain the informal model components automatically. This is useful to promote modifiability and traceability.

We do not believe that integrated methods alone will make the use of formal specification widespread. Other pragmatic issues must be taken into consideration such as training and a change in the culture of the organisation. However we are optimistic that methods like the one described here can contribute to the effective incorporation of formal specification by industry.

REFERENCES

Araújo, J. 1996. *Metamorphosis: an Object-Oriented Requirements Specification Method*, PhD Thesis, University of Lancaster, UK.

Braek, R. and Haugen, O., et al, 1996. *SISU Integrated Methodology*, SISU report L-2001-7, SISU, Norway.

Booch, G., Rumbaugh, J. and Jacobson, I., 1998. *The Unified Modeling Language User Guide*, Addison-Wesley.

Clark, R., and Moreira, A., 1997. Constructing Formal Specifications from Informal Requirements, *Proc. Software Tecnology and Engineering Practice*, IEEE Computer Society.

Duke, D., King, P., Rose, G.A., Smith, G., 1991. *The Object-Z Specification Language*, version 1, Technical Report 91-1, Department of Computing Science, University of Queensland, Australia.

Hayes, I., 1987. editor: *Specification Case Studies*. International Series in Computer Science, Prentice-Hall.

Jacobson, I., 1992. *Object-Oriented Software Engineering – a Use Case Driven Approach*, Addison-Wesley, Reading Massachusetts.

Kotonya, G. and Sommerville, I., 1996. Requirements Engineering with Viewpoints, *Software Engineering Journal*, 11(1), 5-18.

Kuusela, J. and Kenttunen, E., 1993. Integrating SDL and Object-Oriented Analysis Through OMT/SDL, *SDL'93*, 89-103, North Holland.

Manna, Z. and Pnuelli, A., 1992. *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag.

Meyer, B., 1998. *Object-Oriented Software Construction,* Prentice-Hall.

Moreira, A. and R. Clark, R., 1996. Adding Rigour to Object-Oriented Analysis, *Software Engineering Journal*, 11(5): 270-280.

Nuseibeh, B., Kramer, J. and Finkelstein, A., 1994. A Framework for Expressing the Relationships between Multiple Views in Requirements Specification, *IEEE Transactions on Software Engineering*, 20(10), 760-773.

Reed, R., 1996. Methodology for Real Time Systems, *Computer Networks and ISDN Systems*, 28(12), 1685-1701.

Rumbaugh, J. et al, 1991. *Object-oriented Modeling and Design*, Prentice Hall.

Sinclair, D., Clyinch, G. and Stone, B., 1996. An Object-Oriented Methodology from Requirements to Validation, *Proc. OOIS'95*, Springer-Verlag.

Spivey, J.M., 1992. *The Z Notation: A Reference Manual*. 2nd edition, Prentice-Hall.

Telelogic, 1998. SDT 3.4 Methodology Guidelines Part 1: The SOMT Method.