2000

Specifying the Behavior of UML Collaborations Using Object-Z

Joao Araujo
Universidade Nova de Lisboa, ja@di.fct.unl.pt

Ana Moreira
Universidade Nova de Lisboa, amm@di.fct.unl.pt

Specifying the Behaviour of UML Collaborations Using Object-Z

João Araújo and Ana Moreira

Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2825 Monte da Caparica, PORTUGAL, {ja|amm}@di.fct.unl.pt

Abstract

UML is a standard modelling language that is able to specify a wide range of object-oriented concepts. Among them, we have collaborations, that serve to realise use cases, a powerful abstraction concept. The behaviour part of a collaboration is rendered using collaboration diagrams. However, the lack of formalisation compromises the precision of the specification. By using formal description techniques, such as Object-Z, we can reason about the requirements and identify ambiguities and inconsistencies earlier in the development process. In general, we can say that formalisation helps obtaining a more reliable system. Our aim is to formalise collaborations Object-Z class schemas. This is accomplished by proposing an integrated formal process.

Keywords: Object-oriented analysis, collaboration, Object-Z, UML

1. Introduction

A collaboration, in UML (Booch 1998), is "a society of classes, interfaces and other elements that work together to provide some cooperative behaviour". This consists of a structural part and a behavioural part. A class diagram typically specifies the structural part. The behavioural part is rendered using one or more interaction diagrams, i.e. sequence and collaboration diagrams. A sequence diagram shows the time ordering of messages exchanged between the objects involved, and the collaboration diagram emphasises the structural relationships among the objects involved. Both are semantically equivalent.

Collaborations serve to the realisation of use cases. 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. Software developers are easily seduced by the simplicity and potentiality of use cases; they claim that use cases are an easily understood technique for capturing requirements.

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 the behavioural part of UML collaborations formally, starting from the specification of use cases. We will adopt collaboration diagrams to represent the behavioural part of a collaboration. This has the advantage over sequence diagrams because the links among the objects helps specifying the class diagram of the structural part of the collaboration.

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. 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. Hammond (Hammond 1994) integrates the Shlaer-Mellor (Shlaer 1992) method with Z (Spivey 1992). Hammond uses Hall's recommendations (Hall 1994) for writing object-oriented specifications in Z. France *et* al. (France 1997) present an environment that supports the integration of Fusion (Coleman 1994) and Z. These approaches do not use a truly object-oriented formal language, compromising the homogeneity and readability of the specification.

Other works adopt object-oriented formal languages (Duke 1991; Z.100 1994). Kuusela (Kuusela 1993) use SDL, but only for the design phase. For the analysis phase they use OMT (Rumbaugh 1991). Lano (Lano 1995) uses OMT and Booch (Booch 1994) combined with the formal languages VDM++ (Durr 1993) and Z++ (Lano 1991). Moreira and Clark developed ROOA (Moreira 1996) to build a formal and executable object-oriented specification from informal requirements using SDL (Z.100 1994) or LOTOS (ISO 1988). Araújo and Sawyer developed Metamorphosis (Araújo 1998) to combine an object-oriented model with Object-Z (Duke 1991). However, none of them considers formalising UML's object model (Booch 1998).

There is some work done that uses Object-Z to formalise UML model components as, for example, class diagrams (Kim 1999), and persistence, class views and excluding classes (Araújo 1999).

The work that has been done by Övergaard (Övergaard 1999) presents a formal definition of the collaboration construct in the UML. Our work concentrates on the formalisation of collaborations of an application model.

3. Object-Z

Object-Z is a well-known extension of Z to incorporate object-oriented concepts (Meyer 1998). Object-Z has been used in many real applications, such as 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 (see Figure 1).

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;
- 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.

Figure 1. Object-Z class schema

```
┌─ ClassName [generic parameters] ───
│ visibility list
│ inherited classes
│ type definitions
│ constant definitions
│ state schema
│ initial state schema
│ operation schemas
│ history invariant
└────────────────────────────────────
```

This last component was determinant for the choice of this language as it provides a straightforward mechanism to represent the dynamic behaviour, that fits perfectly to our purposes.

4. Overview of the Process

In this paper we propose a process that derives a formal object-oriented specification for the behaviour part of collaborations, using Object-Z, starting from a use case model. This process is shown in Figure 2.

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 collaboration 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 and respective collaborations, specify the collaboration diagrams for each collaboration and from here generate Object-Z class schemas. Each collaboration, translated into collaboration diagrams, offers partial views of several objects. These views when integrated show the complete functionality of the system.

The formal specification 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 collaboration diagram. The formal specification presented here is incremental, and we do not have to formalise the classes beforehand to formalise the collaboration diagram, their specification can be done *a posteriori*.

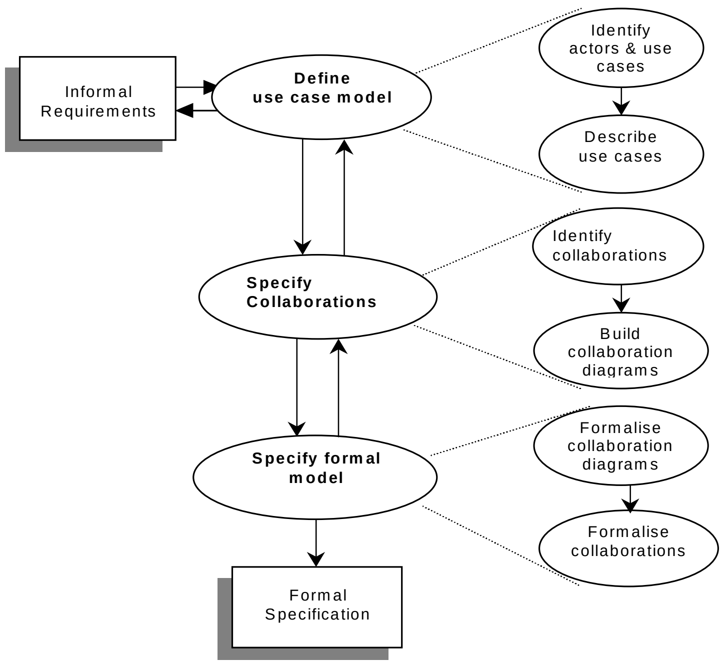


Figure 2. The process to formalise the behaviour of collaborations

As we understand more of the requirements, we can introduce either more detailed information in a use case, or add new use cases and the respective collaborations to our system. This new information can either be added to existing collaboration 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). This is a simplified version of the real system.

"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.

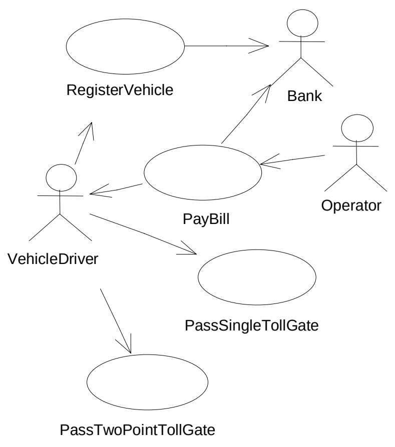
When an authorised vehicle passes through a green lane, a green light is turned on, and the amount being debited is displayed. If an unauthorised vehicle passes through it, a yellow light is turned on and a camera takes a photo of the plate (that will be used to fine the owner of the vehicle).

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

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 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.

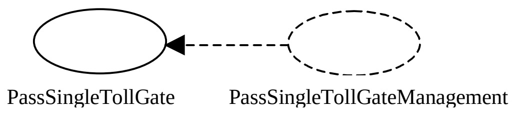
The use cases are:

- 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 valid the light is turned green, and the amount to be paid is calculated and displayed; if the gizmo is not valid, 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 valid, 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.

5.3 Specify collaborations

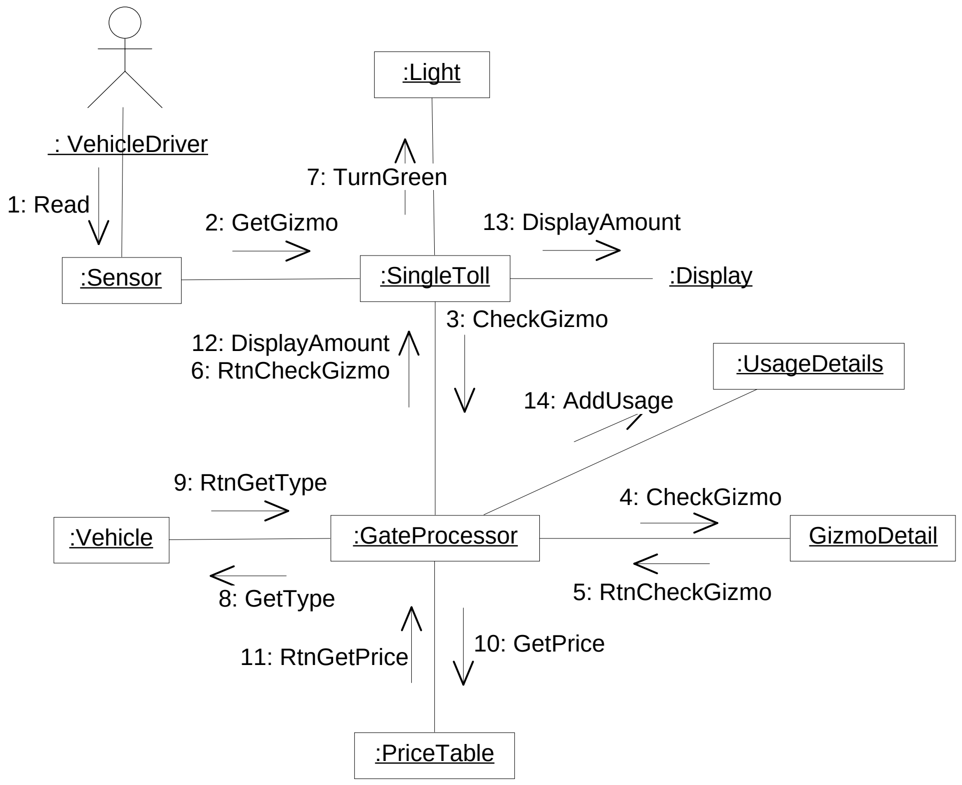
Collaborations realise uses cases, through a realisation relationship (represented by a dashed arrow). To exemplify this, we choose the use case *PassSingleTollGate*, which deals with two situations: authorised vehicles and non-authorised vehicles. The associated collaboration for that is *PassSingleTollGateManagement*. 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 toll gate



In the collaboration diagram, objects are shown as icons whose naming scheme takes the form *objectName:ClassName*. Arrows represent the messages sent in a collaboration and their sequence is indicated by numbering them. Conditions can be specified between square brackets and before the names of the messages. We use separate diagrams for each scenario. Figure 5 shows the collaboration diagram for authorised vehicles, passing a single toll (*PassSingleTollGateOk*).

Figure 5. Collaboration diagram depicting an authorised vehicle passing a single toll gate



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

5.4 Specify the formal model

Our formal object-oriented specification is centred on the behaviour of the collaborations. The rules defined to generate an Object-Z specification from the associated collaboration diagrams 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):

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. Collaboration diagrams show the links, the message passing, and the synchronisation among objects, which can naturally be expressed by temporal logic. Therefore, it is practicable then to translate collaboration diagrams into history invariants.

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

1. A collaboration diagram can be mapped into Object-Z as a class schema where its label is derived from the collaboration diagram name defined in the respective template. In the example, the class schema name generated is *PassSingleTollGateOK*.
2. The objects that participate in the collaboration 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 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 collaboration diagram.
4. The message passing of the collaboration diagram and its ordering is converted into a history invariant that is expressed by a temporal logic formula.
5. Each message, in a collaboration diagram, is passed from a sender object to a receiver object, can have an associated condition and has an sequence 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 (**op** = $o_{i+1}.m_i$) or ($condition_i \wedge$ **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
   - $\alpha_j \rightarrow \Diamond(\alpha_{j+1} \rightarrow \dots \Diamond(\alpha_{n-1} \rightarrow \Diamond\alpha_n)\dots)$ where $1<j \leq n$.

The Object-Z class schema of the collaboration diagram *PassSingleTollGateOK*, illustrated in Figure 6, is obtained by applying the rules above.

Having specified and formalised collaboration diagrams for each collaboration identified, we are able to formalise the whole behaviour of the collaboration itself. Each collaboration can be mapped into a class schema following the rules below:

1. The name of the class schema has the same name of the collaboration. For example, in our case the class schema name is *PassSingleTollGateManagement.*
2. Each collaboration class schema inherits all the class schemas derived from the respective collaboration diagrams.
3. Explicit renaming of variables is the responsibility of the specifier, if this is necessary.

In the example, the collaboration *PassSingleTollGateManagement* generates the class schema shown in Figure 7, by applying the rules above.

Figure 6. Class schema of *PassSingleTollGateOK*

```
┌ PassSingleTollgateOK ─────────────────────────
│ ┌──────────────────────────────────────────
│ │ s: Sensor
│ │ st: SingleToll
│ │ gp: GateProcessor
│ │ gd: GizmoDetail
│ │ v: Vehicle
│ │ pt: PriceTable
│ │ ud: UsageDetail
│ │ l: Light
│ │ d: Display
│ └──────────────────────────────────────────
│ ┌ Init ─────────────────────────────────────
│ │ s.Init ∧ st.Init ∧ gp.Init ∧ gd.Init ∧ pt.Init
│ │ ∧ ud.Init ∧ l.Init ∧ d.Init
│ └──────────────────────────────────────────
├─────────────────────────────
│ □ (op = s.Read →
│    ◊ (op = st.GetGizmo →
│    ◊ (op = gp.CheckGizmo →
│    ◊ (op = gd.CheckGizmo →
│    ◊ (op = gp.RtnCheckGizmo →
│    ◊ (op = st.RtnCheckGizmo →
│    ◊ (op = l.TurnGreen) →
│    ◊ (op = v.GetType →
│    ◊ (op = gp.RtnGetType →
│    ◊ (op = ud.GetPrice →
│    ◊ (op = gp.RtnGetPrice→
│    ◊ (op = st.DisplayAmount →
│    ◊ (op = d. DisplayAmount →
│    ◊ (op = ud.AddUsage)))))))))))))
└──────────────────────────────────────────────
```

Figure 7. Class schema of the behaviour of the collaboration *PassSingleTollGateManagement*

```
┌ PassSingleTollgateManagement ───────
│
│ PassSingleTollGateOK,
│ PassSingleTollgateNotOK
└─────────────────────────────────────
```

With the objects, their links, messages and services identified we can build a class diagram to represent the structural part of the collaboration. 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 collaborations (and collaboration diagrams) into an object-oriented formal notation (Object-Z). The work described here has the exclusivity to introduce rules to transform the collaborations into Object-Z's class schemas. Translating collaborations into object-Z specifications provides a sound mechanism to reason about the semantics of the collaborations. 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 process to handle concurrency from the point of view of an object that receives simultaneously the same message from different senders, and from an object that broadcast a message to different objects. 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.

References

Araújo, J. and Sawyer, P. "Integrating Object-Oriented Analysis and Formal Specification," *Journal of Brazilian Computer Society* (5:1), 1998.

Araújo, J. and Moreira, A., Sawyer, P. "Specifying Persistence, Class Views and Excluding Classes for UML," *12th International Conference on Software Engineering*, Paris, France, December 1999.

Booch, G. *Object-Oriented Design with Applications*, Benjamim-Cummings, Menlo Park, California, 1994.

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

Clark, R. and Moreira, A. Constructing Formal Specifications from Informal Requirements, *Software Tecnology and Engineering Practice*, IEEE Computer Society, Los Alamitos, California, July 1997, pp. 68-75.

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F. and Jeremaes, P. *Object-Oriented Development - The Fusion Method*, Prentice-Hall, Englewood Cliffs, New Jersey, 1994.

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.

Durr, E. "VDM++: A Formal Description Language for Object-Oriented Designs," IEEE CompEuro, 1991, pp. 214-219.

France, R., Bruel, J.-M. and Larrondo-Petrie, M. M. "An Integrated Object-Oriented and Formal Modeling Environment," *JOOP* (10:7), 1997.

Hall, A. "Specifying and Interpreting Class Hierarchies in Z," *8th Z User Workshop*, Cambridge, U.K., 1994.

Hammond, J. "Producing Z Specifications From Object-Oriented Analysis," *8th Z User Workshop*, Cambridge, U.K., 1994, pp. 317-336.

ISO, Information Processing Systems - Open Systems Interconnection - LOTOS: A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, International Standard 8807, ISO, 1988.

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

Kim, S. and Carrington, D., Ed. "Formalizing the UML Class Diagram Using Object-Z," *UML'99: Beyond the Standard, Lecture Notes in Computer Science*, R. France and B. Rumpe (eds.), vol. 1723, Springer-Verlag, Berlin, Germany, 1999, pp. 83-98.

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

Lano, K. "Z++, An Object-Oriented Extension to Z," *Workshop in Computing*, J. Nicholls (ed.), Springer-Verlag, Oxford, U.K, 1991.

Lano, K. *Formal Object-Oriented Specification Development*, Springer-Verlag, Heidel-berg, Germany, 1995.

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

Meyer, B. *Object-Oriented Software Construction*, Prentice-Hall, Upper Saddle River, New Jersey, 1998.

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

Övergaard, G. A Formal Approach to Collaborations in the Unified Modeling Language," *UML'99: Beyond the Standard, Lecture Notes in Computer Science*, R. France and B. Rumpe (eds.), vol. 1723, Springer-Verlag, Heidelberg, Germany, 1999, pp. 99-115.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

Shlaer, S. and Mellor, S. *Object Lifecycles: Modelling the World in States*, Yourdon Press, Englewood Cliffs, New Jersey, 1992.

Spivey, J. M. *The Z Notation: A Reference Manual*, Prentice-Hall, Hemel Hempstead, U.K., 1992.

Z.100. Specification and Description Language SDL, ITU-T, 1994.