Rigorous Object-Oriented Analysis

Ana Maria Dinis Moreira

Department of Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA

This thesis has been submitted to the University of Stirling in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

August 1994

Abstract

Object-oriented methods for analysis, design and programming are commonly used by software engineers. Formal description techniques, however, are mainly used in a research environment. We have investigated how rigour can be introduced into the analysis phase of the software development process by combining object-oriented analysis (OOA) methods with formal description techniques. The main topics of this investigation are a formal interpretation of the OOA constructs using LOTOS, a mathematical definition of the basic OOA concepts using a simple denotational semantics and a new method for object-oriented analysis that we call the Rigorous Object-Oriented Analysis method (ROOA).

The LOTOS interpretation of the OOA concepts is an intrinsic part of the ROOA method. It was designed in such a way that software engineers with no experience in LOTOS, can still use ROOA.

The denotational semantics of the concepts of object-oriented analysis illuminates the formal syntactic transformations within ROOA and guarantees that the basic object-oriented concepts can be understood independently of the specification language we use.

The ROOA method starts from a set of informal requirements and an object model and produces a formal object-oriented analysis model that acts as a requirements specification. The resulting formal model integrates the static, dynamic and functional properties of a system in contrast to existing OOA methods which are informal and produce three separate models that are difficult to integrate and keep consistent. ROOA provides a systematic development process, by proposing a set of rules to be followed during the analysis phase. During the application of these rules, auxiliary structures are created to help in tracing the requirements through to the final formal model.

As LOTOS produces executable specifications, prototyping can be used to check the conformance of the specification against the original requirements and to detect inconsistencies, omissions and ambiguities early in the development process.

Declaration

I hereby declare that this thesis has been composed by myself, that the work reported therein has not been presented for any university degree before, and the ideas that I do not attribute to others are due to myself.

Ana Maria Dinis Moreira
August 1994

Acknowledgements

Dr. Robert Clark was my supervisor in this thesis. I was very lucky to have the opportunity to work with him. It was Dr. Clark who suggested the ideas that would grow into the work I am presenting now. I have to thank him for that. But, more than that, I have to thank him for being always ready to read and comment on what I was writing, be it notes, examples, technical papers, and finally this thesis, and for being always available for our discussions over these three years. Thank you, Bob, I hope you may find that the time you spent helping me was not wasted.

Prof. Pedro Guerreiro is my colleague and dear friend in Lisbon. I learnt with him how to program, I wrote my M.Sc. thesis under his supervision and worked with him in the Departamento de Informática of Universidade Nova de Lisboa for four years. He is the person I have to thank in the first place for having the opportunity to come to Stirling. On top of all this, I owe him all the time he spent reading my technical reports, my papers and this thesis. Thank you, Pedro, for being always there when needed.

Dr. Peter Ladkin was my colleague at the Department, here at Stirling. He took an interest in my work, and become acquainted with it, although this was somewhat of a new subject for him. In the process, we had many discussions that helped me clarify my own ideas. He taught me many things about denotational semantics, and we worked together on the ideas that led to the theory presented in Chapter 5, in particular the ideas presented in Section 5.3 and Section 5.5.3. Thank you, Peter, without your help this thesis would have taken much longer to complete.

Prof. Kenneth Turner and Mr. Charles Rattray also helped me to accomplish this thesis. They were always available to comment on my papers and technical reports during these years. Also, Mr. Steve Marsh helped by reading an earlier draft of this thesis. Thank you, Ken, Chic and Steve for your patience with me.

Lim Pink wrote her M.Sc. dissertation on an assessment to the ROOA method. This gave me the insight on how a newcomer can learn and apply ROOA. Thank you, Pink, for being a good friend, and for reading a previous draft of this thesis.

The computer officers in the department had to endure my complaints about software that did not seem to work, e-mails that did not arrive, workstations behaving strangely. Thank you, Sam, Graham and Catherine, for putting up with my questions.

My friends in Stirling helped in the moments I felt lonely and far from home. Thank you, Amelia, António, Ashley, Claire, Dave, Gary, Gill, Julie, Manfred, Mike, Paul, Pink, Steve and Toya! Thank you to all people in the Department for making me feel welcome. Working in Stirling was a great experience. I will miss you all.

My parents, Fernando and Lucinda, deserve all my love for everything they always do for me. "Bigada, Papinho e Maminha, pelo vosso carinho, compreensão e apoio."

The Junta Nacional de Investigação Científica e Technológica (JNICT) financed this Ph.D. through Programa CIENCIA. The Department of Computing Science and Mathematics, University of Stirling, provided financial support towards travelling expenses for conferences.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Object-oriented approaches and formal methods have both been proposed as ways of alleviating problems in the development and maintenance of reliable software systems. Object-oriented approaches are gradually becoming more and more accepted in industry, during all phases of software development. Formal methods are also gradually becoming more used in industry, but they are not usually introduced until the design phase. This is because the construction of an initial formal specification during early stages of development is difficult. In fact, little previous work has been done in the area of object-oriented analysis and formal methods.

The starting point in the use of formal methods in the software development process is a formal requirements specification of what the proposed system is to achieve. Once a formal specification has been given, it is possible, at least in theory, to verify a design and eventual implementation with respect to that specification. Two important questions remain, however. How is the initial formal requirements specification created from a set of informal requirements and how can it be validated with respect to those requirements? It is clear that these cannot be formal processes.

The purpose of this thesis is to investigate how formal methods can be used within the context of object-oriented analysis, during the requirements analysis phase of the software life cycle. The result of this investigation is the Rigorous Object-Oriented Analysis (ROOA) method. The ROOA method specifies the required behaviour of a system by constructing a model using the formal description technique LOTOS (Language Of Temporal Ordering Specification) [BB87, ISO88]. As LOTOS has a formal semantics, the model has a precise meaning and can be used as a formal requirements specification of the system's intended behaviour.

1.1 The Scope and Objective of the Thesis

This thesis is primarily concerned with investigating how LOTOS can be used to add rigour to the object-oriented analysis process. To accomplish this, it was necessary to:

- assess the existing object-oriented analysis methods and identify which characteristics would be useful to consider, to extend or to modify;
- study the LOTOS specification language and identify in which way it could be used to model the most common object-oriented concepts;

- analyse how LOTOS could be brought to co-exist with informal object-oriented analysis methods;
- develop a method which would integrate the LOTOS formal language with the existing object-oriented analysis methods;
- propose a formal definition of the basic object-oriented concepts.

The result of this investigation is the ROOA method, together with both a mapping which shows how each object-oriented concept can be specified in LOTOS and a simple denotational semantics formalising basic concepts such as class template, class, object, behaviour, state, attributes and services.

1.2 The Contribution of the Thesis

This section summarises the contributions made by the thesis, which are:

- propose a method which integrates formal specification languages with object-oriented analysis (OOA) methods;
- explore the suitability of the LOTOS language to create an initial formal requirements specification;
- propose a formal interpretation of object-oriented analysis constructs in LOTOS;
- develop a simple denotational semantics describing basic object-oriented concepts.

1.2.1 Integrating Formal Specification Languages with OOA

The major contribution of this thesis is the investigation of how formal methods can be used during the requirements analysis phase to develop object-oriented systems. The advantages of this integration are to bring formal specification languages closer to the everyday work of software engineers, helping them to lose the fear of using formal languages. Also, by using formal methods early in the development life-cycle we can create an initial formal requirements specification which we can then use as the starting point of a more formal development.

The result of this study is the ROOA method. ROOA proposes a systematic development process to create an initial formal requirements specification. It brings together two different frameworks which have been previously kept apart: OOA methods and formal specification languages.

Object-oriented analysis methods such as those by Coad and Yourdon, by Jacobson, by Rumbaugh *et al.*, and by Shlaer and Mellor create an informal analysis model or set of models [CY91a, Jac92, RBP +91, SM92]. The *object model*, based on an extension to entity-relationship diagrams, describes the static properties of a system while the *dynamic model*, normally expressed in terms of state transition diagrams, describes its behaviour. Some methods, such as those by Rumbaugh *et al.* and by Shlaer and Mellor, also propose a *functional model*, which uses data flow diagrams to describe the operations in the object

model and the actions in the dynamic model [RBP +91, SM92]. In most methods, the object model is central with the dynamic and functional models being of lesser importance.

Entities in the real world exist concurrently. A major advantage of the object-oriented approach is that it supports the direct modelling of real world entities as a set of autonomous objects which communicate with one another by sending messages. An object-oriented analysis model should therefore represent the requirements as a set of communicating concurrent objects even when the eventual implementation is to be sequential. A formal language used to formalise the analysis model should therefore support parallelism.

LOTOS is composed of a process algebra and abstract data types and it does not directly incorporate any object-oriented constructs. This was not one of the goals of its designers, and understandably so: when LOTOS was designed, object technology was not a mainstream concern. However, the language is suitable for writing specifications in an object-oriented style [Rud92]. Discovering ways to accomplish this was one of our main concerns while developing ROOA.

LOTOS directly supports encapsulation, abstraction and information hiding. Concurrent objects may be modelled as process instances, composed by using the parallel operators, and message passing is modelled as two processes synchronising on an event. This straightforward mapping makes LOTOS capable of representing a system as a set of communicating concurrent objects.

The ROOA method shows how LOTOS can be integrated with object-oriented analysis methods. ROOA complements existing object-oriented analysis methods (such as those by Rumbaugh *et al.*, Coad and Yourdon and Shlaer-Mellor), enabling precision and formality in development where required, for example in safety-sensitive systems [CY91a, RBP +91, SM89].

Producing an object model from a set of informal requirements is a complex process. OOA methods propose strategies for the identification of objects and their attributes, services and relationships so that a suitable object model can be created. Most OOA methods are much better at describing the static aspects of a problem than they are at describing the expected dynamic behaviour. The ROOA method builds on the object model produced by any of the informal OOA methods and on the dynamic and functional properties given in the informal requirements.

The model produced by ROOA integrates the static, dynamic and functional models, unlike informal object-oriented analysis methods, such as the one by Rumbaugh *et al.*, which create three separate models [RBP +91]. It is primarily a dynamic model, but it keeps the structure of the static object model. The object communication table and the object communication diagram are intermediate structures which help us in the difficult transformation from the static to the dynamic model and in tracing the requirements through to the final formal model.

An important part of the ROOA method is to give a formal interpretation in LOTOS of object-oriented analysis constructs such as: class templates, objects, inheritance, relationships between objects, message passing between objects, aggregates, and subsystems.

As a LOTOS specification is executable, we use prototyping to help the developers to understand the user requirements, to discover omissions, contradictions, ambiguities or inconsistencies early in the development process, to validate the resulting specification

against those requirements, and at the same time, to help the users to evaluate their own needs. A set of tools, such as syntax checkers, semantic checkers and simulators, is available with LOTOS. The SMILE simulator supports value generation and allows symbolic execution of a specification where a set of possible values is used rather than particular values [EW93]. In the analysis phase, non-determinism can be exploited to model behaviour so that premature design decisions are not made.

ROOA uses a stepwise refinement approach for the development and for validation of the specification against the requirements. The development process is iterative and parts of the method can be re-applied to subsystems. Different objects can be represented at different levels of abstraction and the model can be refined incrementally.

1.2.2 Suitability of the LOTOS Specification Language

Our goal was to introduce rigour to the object-oriented analysis process by creating an initial formal specification, and, on the other hand, to develop a method which could be given to software engineers to use in real projects. To satisfy this goal, the formal specification language chosen has to incorporate a certain number of characteristics. For example, the language should be well-established and standardised, if possible; it had to be able to produce specifications in an object-oriented style; it had to be supported by tools which then could be used for prototyping; it had to incorporate concurrency, since it is important, in the analysis phase, to see the resulting specification as a set of concurrent objects.

LOTOS satisfies these requirements.

1.2.3 Formal Interpretation of the OOA Constructs in LOTOS

Currently, ROOA offers a mapping between each construct used by most of the existing object-oriented analysis methods and LOTOS. This is an important part of our method, as it helps software engineers less skilled in formal specification languages to use ROOA.

Among the OOA concepts studied, we would like to highlight the contribution of the thesis towards the understanding of aggregates. Aggregation has been studied for some years, but even so there is not a general agreement on the exact meaning of the concept. We devoted a long time to review other authors' views on aggregates and decided to propose our own idea, which is more generic and closer to reality, we believe. We describe two main kinds of aggregates: aggregates with hidden components and aggregates with shared components. We indicate the set of properties which each kind of aggregate should have. Finally, we show how aggregates can be specified in LOTOS.

1.2.4 Denotational Semantics for OOA Models

Having developed the ROOA method and shown how each OOA construct could be specified in LOTOS, it was important to abstract from LOTOS and describe our views of each concept in a mathematical form so that it would be better understood by others.

As a result, we developed a simple denotational semantics which describes, in terms of tuples and functions, the basic object-oriented concepts. During this study it was important to separate the OOA concepts into two kinds: those which appear in the specification and those which only exist when an executable specification runs. This led us to decide that while class templates are concepts which exist in the specification, objects only exist

while a specification is executing. By using the principle of substitution and the notion of (logical) individual variables, we show how objects can be derived from generic descriptions such as class templates. This implies deriving the behaviour of objects from behavioural expressions in class templates.

1.3 Related Work

Clark, in earlier work, proposed an object-based style to write LOTOS specifications [Cla91, Cla92a, Cla92b, CJ92]. There are three main differences between our work and Clark's. First, he was concerned with the design and implementation phases of the software development, while we are concerned with the analysis phase. Second, he proposed an *object-based* style to write LOTOS specifications, while we are interested in finding an *object-oriented* style where the concepts of inheritance, class template, class (object generator), object, aggregation and static relationships between objects, for example, play an important role. Clark had already proposed ways to model classes as LOTOS process definitions, objects as process instances and communication between objects as a pair of LOTOS processes synchronising on an event. Our work goes further and extends the formalisation of object-based concepts to include inheritance and the concepts used by OOA methods, integrating also some of the concepts proposed by the Open Distributed Processing (ODP) model. Third, together with the object-oriented style we have developed to write LOTOS specifications, we are proposing a complete method to create an initial object-oriented analysis requirements specification and we integrate our ideas within existing OOA methods. This has no counterpart in Clark's previous work.

Many others have been proposing ways to specify object-oriented concepts and LOTOS [Bla89, CRS90, May89, Rud92]. The focus of their work is different from ours, as they are usually concerned with extending LOTOS to incorporate object-oriented concepts. This is the case with Rudkin's work where he proposes extending LOTOS to incorporate inheritance [Rud92]. We have adopted the ISO standard LOTOS language, without considering any of its extensions.

Jungclaus and his colleagues, while integrating formality with object-oriented concepts, developed a formal object-oriented specification language called TROLL [JSHS91]. Our work differs from theirs in that we have developed a method which integrates a standard formal specification language with existing OOA methods. (Other specification languages which incorporate object-oriented concepts are Object Z [SBC92], OBJ3 [GKK $^{+}$88] and ObjLog [Bri94].)

Bahsoun *et al.* [BMS93] proposed ways to formalise object-oriented concepts using Lamport's TLA logic [Lam94], while formalisations using universal algebra and category theory have been reported by Ehrich and his colleagues [EGS93, EGS91, ESS89]. However these approaches are different from ours (as we further explain in Chapter 5) as we give primacy to the concepts as they exist in the OOA methods.

Jones [Jon93], Laorakpong and Saeki [LS93], and Cusack and Lai [CL91] have been working on formalising object-oriented specifications. These approaches introduce a new methodology, but they do not seem to build on existing practical approaches, as ROOA does.

The team developing the ODP model [ISO94] are proposing ways to model the concepts of class template, object, class and service in specification languages such as LOTOS, Z

and SDL. However, their goal is to produce a model for telecommunication networks, not to add rigour to OOA methods, which is the primary objective of ROOA.

The projects RAISE [GHH +92] and SPECS [Gen92] were proposed to formalise requirements analysis. Another author, Li, also proposes a way to formalise requirements [Li93]. However while we were interested in bringing formality to the existing object-oriented analysis process, they were proposing completely different approaches, not taking into consideration existing OOA methods.

Hedlund reported some work on integrating LOTOS with object-oriented development methods [Hed93]. He does not, however, deal with the analysis phase and his study is restricted to the Jacobson's method [Jac92]. Our approach has similarities with that of Zave [Zav91], although a major feature of ROOA is its integration with object-oriented analysis methods.

1.4 The Structure of the Thesis

This thesis has nine chapters and four appendices. Chapter 1 is this introduction. We now briefly present the remaining chapters and the appendices.

Chapter 2. *Formal Object-Oriented Specifications.* This chapter gives the software engineering background to the ROOA development. It starts with a brief introduction to software development life-cycle models, emphasising the relationship between the analysis and design phases. It then discusses the object-oriented paradigm, examining in particular object-oriented analysis methods and their relation to object-oriented design. Finally, it identifies the different kinds of specifications, groups them according to their properties, analyses the benefits derived from combining object-oriented analysis methods with formal methods, states the reasons for LOTOS as the formal description technique to be used within ROOA, and gives a general overview of prototyping, emphasising its use during the analysis phase.

Chapter 3. *Modelling Fundamental OOA Concepts in LOTOS.* This chapter gives an informal definition of each of the most common object-oriented analysis concepts, such as inheritance, class templates, objects, services, attributes, relationships and message connections between objects. We use the problem of specifying an automated banking system as an example to show how to model each of these concepts in LOTOS, including inheritance.

Chapter 4. *Further Concepts: Complex Objects.* This chapter focuses on complex objects, usually known as aggregates. It briefly describes how aggregates are understood by other authors, then presents our own views by classifying aggregates according to the properties of their components, and by defining the properties we believe aggregates should have. Finally, it uses a simple example to show how ROOA models each kind of aggregate in LOTOS.

Chapter 5. *Formalising Object-Oriented Analysis With LOTOS.* This chapter provides a simple denotational semantics to describe the basic object-oriented concepts in terms of simple mathematical formalisms such as tuples, sets and functions. It divides the concepts into two types: the ones which appear in the specification, such as class templates and services, and the ones which only come into existence in the running system, such as

objects. The main focus here is to explain how the behaviour of objects at run-time is specified through the generic behaviour description given in the class template.

Chapter 6. *The Rigorous Object-Oriented Analysis Method.* This chapter describes the ROOA method. It presents ROOA in the context of the software life cycle, showing how it interacts with the requirements capture and with the design phase. It discusses the development model adopted within ROOA. The main part of this chapter describes in detail each task and subtask of ROOA. We illustrate this description by using the automated banking system, whose requirements were presented in Chapter 3. Some intermediate structures are used and explained while applying ROOA to this example.

The two primary intermediate structures used are the object communication table and object communication diagram. They are used to develop the final object-oriented analysis LOTOS model and to help trace through the original requirements to the LOTOS specification. An algorithm is presented that helps us to create the object communication diagrams.

Finally, the chapter discusses the various documents created during application of the ROOA method.

Chapter 7. *The Design Rationale of ROOA.* This chapter explores the problems we found and the solutions tried during the development of ROOA. It justifies the solutions we adopted, and it also discusses the reasons why we discarded some of the intermediate solutions.

Chapter 8. *Assessment of the ROOA Method.* This chapter discusses the need for a rigorous object-oriented analysis method, assesses the ROOA method, highlighting its strengths and weaknesses, examines the suitability of LOTOS as the chosen specification language, and finally, reviews the ROOA method while applied to case studies by ourselves and by others.

Chapter 9. *Conclusions and Prospects.* This chapter states the conclusions of this work and identifies possibilities for further development. Finally, it identifies the transformations in ROOA which can be performed automatically and outlines the tools which could be developed to support ROOA.

Appendix A. *LOTOS Overview.* This appendix contains a brief introduction to the LOTOS specification language. It describes processes, abstract data types and the structure of a LOTOS specification, and presents the syntax of the most important LOTOS operators.

Appendix B. *How to Specify Object Identifiers: Additions to the Library.* This appendix contains the specification of the ADTs which we introduced to the existing LOTOS libraries. These ADTs were created to deal with object identifiers.

Appendix C. *Publications.* This appendix shows a list of our technical reports and publications, in conferences and journals, under the topics: object-oriented methods, and formal description techniques.

Appendix D. *Acronyms.* This appendix lists the abbreviations used in this thesis. We tried to keep acronyms to a minimum, to make reading easier, and more pleasant, we hope.

1.5 The Topics of the Thesis

The thesis discusses three main topics: the ROOA method, the philosophy behind ROOA and a simple denotational semantics. To learn about the ROOA method, Chapter 3, Chapter 4 and Chapter 6 should be read in sequence. To understand the philosophy behind the method, its strengths and weaknesses, Chapter 7 and Chapter 8 should then be read.

For those interested in a simple denotational semantics to define the basic object-oriented concepts, we advise that Chapter 5 should be read after Chapter 3.

Chapter 2

Formal Object-Oriented Specifications

2.1 Software Development

Developing an efficient and reliable software system is a difficult task. As Booch says [Boo87]:

> Developing software systems is an activity that demands much intellectual capacity. Completing an efficient, reliable, maintainable, and understandable system on time is often an even more Herculean task, especially in the case of large, real-time programming projects.

When software systems were small by today's standards, programmers believed that building software was synonymous to writing programs, sometimes in a very artistic way. Today, software systems are much more complex and so software engineers are required to use more skills than just programming artistry when developing the system. They need techniques, methods, and tools that help them progress through a set of required stages.

Since the sixties, many techniques, methods and tools have been produced to support software development. However, the subject remains an active area of research, since the systems we want to develop are still growing in complexity and size. Furthermore, the clients who want those systems are more and more demanding.

In this thesis we are especially interested in two areas of the software technology evolution: object-oriented analysis and formal methods.

2.1.1 Software Life-Cycle

The software life-cycle begins when a software product is conceived and ends when the product is retired [IEE91]. The starting point is usually the set of user requirements. Part of these requirements define what the system must achieve (these are called *functional requirements*). Other part of the user requirements deal with constraints such as efficiency, available hardware, response-time, etc. We refer to these constraints as *non-functional requirements*.

The software life-cycle consists of several phases. The standard approach considers five phases: analysis, design, implementation, testing and maintenance. The boundary between each phase is not always clear. A simple definition of each phase could be:

- Analysis is the study of a problem, prior to taking some action [DeM79]. During this phase we work out the user requirements in order to establish the properties the system should possess. As everybody says, analysis defines *what* the system must do. The result of this phase is the *requirements specification*.

- Design is concerned with *how* the system is going to accomplish what was defined during analysis. In the design we concentrate on determining which are the useful components (data structures and functions) and on how they are going to be implemented. The requirements specification obtained at the end of the analysis phase is the starting point for the design phase. However, it is not always clear where analysis finishes and design starts.

- Implementation is the process of transforming the results of the design phase into instructions for the computer, by using a programming language. Implementation should deliver programs that are correct and efficient.

- Testing "demonstrates" that the programs written in the implementation phase satisfy the requirements specification and correctly transform the input data. During this phase we correct implementation errors and eliminate unexpected program states.

- Maintenance represents all the operations that are necessary to ensure the continuous working of the system. This is an important phase even when the software system is working according to the user requirements. We must accept continuous and constant evolution as a natural law of software. After the system has been delivered, requirements can be modified, the environment in which the system operates can change and an increase in the efficiency and quality of the system can be required.

Our work is concerned with the analysis phase.

2.1.2 Software Development Models

There are several ways to go from one phase to another, that is, there are several life-cycle models. Some of those models are simple, others are more complex.

The original *sequential waterfall* or *stagewise model*, was first presented in [Ben56, Ben83]. However, this model is always applied with feedback, based on the changes proposed by Royce [Roy70]. The five phases are applied sequentially during the development and use of the software product. At the end of each phase we can go back to the previous phase, introduce a change and then propagate the effects of that change. Consequently, we can only step back to an earlier phase to correct errors. This is also called the *iterative waterfall* or *cascade* model. Swartout and Balzer showed why the waterfall model with well defined phases could not work, by showing that the specification and implementation of a system are inevitably intertwined [SB82].

The problems of sequential ordering and strictly separated phases can be overcome by prototyping models, such as the *spiral model* popularised by Boehm [Bel86, Boe88, BB89, BGS84] and the ones in [Fai85, Pet88, Som92]. These models basically have the same phases as the waterfall model, but they permit us to interleave the phases during the development. This follows from Swartout and Balzer. If the implementation phase shows that we need to go back and modify the specification, we reopen the design phase

concurrently with the implementation phase, and perform the necessary changes. Then we propagate the effects of the changes to the implementation, until the design and the implementation come to a point where they match again.

The prototyping models use formalisms for analysis and design phases which lead to executable specifications, promoting prototyping during the early stages of the development. The spiral model has been accepted by the US Air Force as their model for software development [Gre89].

We embrace rapid prototyping in the ROOA development process. We use LOTOS [ISO88] as an executable specification language and the LITE tool [EW93] as the prototyping tool.

2.1.3 Relationship Between Analysis and Design

In every book on the subject, the classic definition for analysis and design is always there: analysis states *what* the system should do, while design states *how* the system accomplishes what was defined during analysis. But, does this tell us exactly what is analysis and what is design? Does it help us to decide when analysis finishes and design starts? As Davis notes, the division into *what* and *how* can be subject to individual perceptions [Dav88]. If we are using functional methods, we can say that in analysis we construct data flow diagrams and entity-relationship diagrams and in design we construct structure charts, apply the third normal form to the data structures and transform the entities in the entity-relationship diagram into data base tables (skeleton tables). Letting the techniques drive us along the development works for some kinds of problems. However, other problems exist where the frontier between analysis and design requires more precision.

We could say that analysis deals with the problem-space while design deals with the solution-space. In an object-oriented view, analysis models the world by identifying the classes and objects that form the vocabulary of the problem domain and design invents the mechanisms that provide the behaviour required by this model [Boo91]. However, deciding when design should start is often a problem. If we start too early, we risk not knowing enough about the problem to make a good design. If we start too late, we risk wasting too much time in doing detailed analysis which can overload the designer with lots of unnecessary information.

The level of abstraction is also important, because we do not want to overspecify the problem. Suppose we have an automatic banking application and we want our system to be able to handle several clients simultaneously. Should the concurrency be shown in analysis, or not? Is the protection required for concurrent access to the data a non-functional requirement? Should non-functional requirements be ignored in the analysis phase and only included in the design, and if so, can we then say that the final document produced during the analysis phase meets the user requirements?

Depending on the method used we might have different levels of abstraction in the specifications. In some methods we can easily avoid having to specify concurrency or data access-protection, in others it might be more difficult. If we specify a system by only defining its external behaviour, rather than by modelling its behaviour, then cases such as the banking example can initially be defined without including parallelism or data access-protection in the model, with the extra non-functional requirements being held informally in a separate document.

When clients write a requirements document, they use their knowledge of the system.

As their knowledge may be limited, the original requirements can be unnecessarily detailed when describing parts of the system they know well, but can have ambiguities, inconsistencies and contradictions, and can omit information about parts that they know less well. Certain kinds of clients like to impose non-functional requirements which mandate implementation of the system in a certain way.

Non-functional requirements should not constrain the analysis. The design might either be restricted or not, depending on what the non-functional requirements look like. The implementation will always be restricted. In other words, non-functional requirements cannot be ignored in the solution, but they should only be considered in the right place.

The question of whether or not a requirements analysis specification document meets the original requirements is fundamental. However, if gaps are found in the original requirements document, they must be corrected. Only then we can assume that the specification meets the original requirements.

An exact definition of *what analysis is* and *what design is* seems to depend on a particular context; every "how" is a solution to a higher level "what" and a "what" may be part of an even higher level "how". That is, as soon as the context changes the scope of analysis and the scope of design also change.

2.2 Object-Oriented Approaches

The concept of class, in software, is not new. Simula 67 was the first programming language to introduce the concept of class as a mechanism to encapsulate data and the related operations [BDMN80, DN66]. The next important object-oriented language was Smalltalk [GK76, GR83].

Wegner gives a definition for the term *object-oriented*. In his opinion, an object-oriented approach must incorporate the concepts of object, class and inheritance (see Figure 2.1) [Weg87].

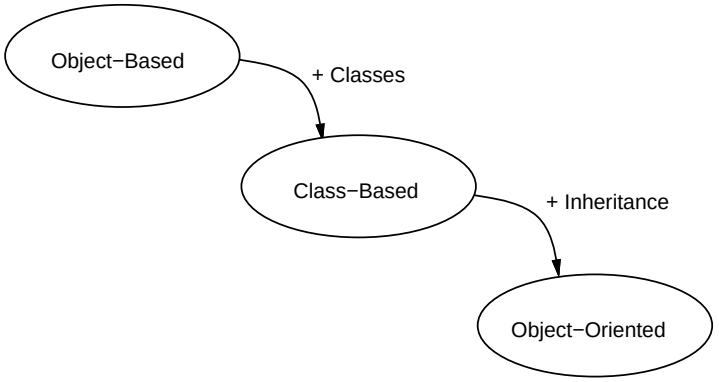

Figure 2.1: Wegner's classification schema

The late eighties and early nineties saw the development of many different object-oriented analysis and design methods [Ber88, Ber89, BGHS91, Boo91, CY91a, CY91b, Col89, CAB+94, Jac92, RG92, RBP+91], as well as an increase in the use of object-oriented and object-based programming languages. Perhaps the most important aspect

is the integration of analysis, design and programming in a single framework, using the same concepts.

There are two major advantages in developing object-based software [BM85]:

- to reduce the total life-cycle software cost by increasing programmer productivity and reducing maintenance costs;
- to implement software systems that resist both accidental and malicious corruption attempts.

The primary goal of modern software development is to encourage the use of software modules to provide specific functionality and whose interactions are through well-defined interfaces. This results in a reduction of the complexity of systems, which, in turn, results in systems which are easier to understand, and easier to build, test and maintain. As a further benefit, the use of modularisation together with abstraction and information hiding results in greater sharing of software and "less reinvention of the wheel", promoting reusability. Object-oriented methods claim to provide these benefits better than other methods [Bha83].

Object-oriented methods are concerned with modelling real world entities as objects. Therefore, they follow one of the most fundamental rules of modern Software Engineering practice, namely "form following function", i.e. making the solution resemble the original problem. Such solutions are inherently easier to test, modify, and debug [Ber88]. This is especially the case with concurrent systems which can be modelled as a set of communicating concurrent objects [Cla90, Cla92b].

2.2.1 Which Approach in the Analysis Phase?

If we decide to use object-oriented methods during design, the question remains as to which approach to choose for analysis, especially if a requirements document does not exist yet. This is because most OOA methods suppose the existence of an informal requirements document (for a further discussion of this problem see [HS93]).

During the early nineties there was much debate about the necessity of having object-oriented analysis methods [Bro91, Fir91, HS92, Shu91, Wal91]. Perhaps because of the expertise already acquired, the powerful support tools (especially CASE tools) created for functional analysis, and the experience already gained with functional methods such as Structured Analysis, some authors have proposed that functional analysis and object-oriented design are compatible [Con89a, Con89b, Shu91, War89]. These authors try to minimise the problem of performing analysis and design with different kinds of approaches by saying that since requirements analysis (saying *what*) and design (saying *how*) have different purposes and serve different masters, it should not be surprising to see different modes of expression.

The reasons for making functional analysis methods compatible with object-oriented design make sense from an managerial point of view, as expertise on functional analysis and automated support tools already exist and so less retraining would be necessary.

There are other authors who believe that functional analysis and object-oriented design are not compatible and state that since the difference between the two approaches is profound, attempts to bend functional methods to object-oriented methods are bound to fail [Bro91, Fir91, Wal91].

Believers in the marriage of the two approaches propose ways to identify objects in a data flow diagram (DFD). We believe that objects which directly represent real world objects are easily identified, but others are usually more arbitrary. While DFDs emphasise processes, object-oriented approaches emphasise objects, which is a very different concept. Also, a DFD process does not correspond to an object-oriented operation. While a process transforms input data into new output data, an object-oriented operation "works" on an abstract object. The data is contained in the object and does not move independently of the object. And so, identifying objects from a DFD seems to be the wrong way to do it, because it leads us to identify objects by means of DFD processes, instead of primarily identifying objects which "own" operations. We cannot say that a Structured Analysis data item is an object, because an object is much more than just data. The use of functional analysis in an otherwise object-oriented approach complicates the tracing of requirements, by forcing the software engineer to look first to a DFD and then switch his or her line of thinking to objects.

The incompatibilities between functional analysis and object-oriented design cause significant problems of quality, productivity and costs [Fir91]. These incompatibilities result from the differences in the abstractions used. Firstly, functional analysis relies primarily upon functional abstraction with minimal data abstraction, while object-oriented design primarily uses object abstraction (but also uses data abstraction, functional abstraction, process abstraction, exception abstraction). Secondly, functional analysis localises (and decomposes) according to functional abstraction, whereas object-oriented design methods localise (and either decompose or compose) according to object abstraction.

We believe that the difficulty of combining effectively two fundamentally different approaches is unnecessary. The best approach for an object-oriented design begins with an object-oriented view of the problem domain created during analysis [Wal91]. If a requirements document does not exist, we can always use methods which integrate the capture of the requirements within the building of the models, such as the method Object Behaviour Analysis proposed by Rubin and Goldberg [RG92]. We believe in OOA methods, but we also know that object-oriented analysis methods are not suited to model all kinds of problems, just as functional methods are not well suited to all kinds of problems.

However, if the problem can be modelled by following the concept of object, an object-oriented analysis method preceding a complementary object-oriented design method promotes a consistency of concept and expression that helps to minimise the traditional errors that occur in the move from analysis to design.

We agree that software engineers must be able to change their preconceived ideas when something new and better appears. But, we understand that this requires hard evidence that the new methods do provide more reliable and cost-effective solutions. With "always arriving new" technologies, each one promising to solve the software crisis, we understand why software engineers might feel sceptical any time a new fashion appears.

In spite of the fact that a great effort is being invested in developing object-oriented technology, a complete acceptance of this paradigm will only be possible after the methods and languages prove their usefulness, and efficient, reliable and powerful supporting tools have been created. Fichman and Kemerer [FK92] state that, although little empirical evidence exists to support many of the claims made in favour of object-orientation, organisations that are able to absorb this radical change may well find themselves in a stronger position than those incapable of making the transition from the traditional functional methods to the object-oriented methods. A fuller discussion of these points is given

in [MC92].

2.2.2 Object-Oriented Analysis Methods

The main goal of an object-oriented analysis (OOA) method is to identify objects and classes which make up the proposed system, to understand the structure and behaviour of each object, to gather in one place all the information relating to a particular object and class and, at the same time, show how the objects in the system interact statically and dynamically.

In general, object-oriented methods share the following set of common tasks:

1. Understand user requirements.
2. Identify and classify objects.
3. Define classes.
4. Identify relationships between objects.
5. Identify inheritance relationships between classes.
6. Construct documentation.

To understand the user requirements we read the initial requirements document and any other source of information which may describe the problem, or part of it. Moreover, we should interview the users or clients of the system so that we can have a better understanding of the problem being studied.

In order to identify objects, several methods [CY91a, RBP +91] recommend we look at nouns, pronouns, noun phrases, adjectival and adverbial phrases in the initial requirements document, while others [RG92] suggest that a better way of identifying objects is to focus on their behaviour. After we identify the objects, we group them into classes.

A class is defined in terms of its static and its dynamic aspects. The static aspect is given by a list of its attributes and services. The dynamic behaviour is usually described by using state transition diagrams, but it plays a secondary role in most methods. The set of state transition diagrams is called the *dynamic model*.

Relationships between objects can be static or dynamic. The static relationships are represented by their names and their cardinality. The dynamic ones are represented by arrows connecting the calling to the called object and are known as *message connections*. These relationships are represented in the *object model* which is supported by a diagram based on Entity-Relationship diagrams [Che76] where enhancements have been introduced to support aggregates, inheritance and message connections. Some methods add *scenarios* [RBP +91] (or *use cases* [Jac92]) to the dynamic model and show the interactions between objects for each scenario by means of an *event trace diagram* [RBP +91] (or *interaction diagram* [Jac92]).

Documentation plays a crucial role when developing software. Several methods have an explicit step to construct it while others leave it implicit.

More recent methods, such as [RBP +91, SM92], also incorporate a *functional model* which uses data flow diagrams to describe the meaning of the services in the object model and the actions in the dynamic model. However, in most methods, the object model plays the central role, with the dynamic and functional models being of less importance.

A major advantage of the object-oriented approach is that, as the concepts used in object-oriented analysis and design are the same, the transition from analysis to design is smoother.

2.2.3 Differentiating OOA from Object-Oriented Design

The use of object-oriented design methods independently of any object-oriented analysis method has created confusion about the responsibilities of analysis and design.

In Section 2.2.1 we discussed the dilemma of whether object-oriented analysis or functional analysis should be used with object-oriented design. In this section we want to clarify the responsibilities of object-oriented analysis and object-oriented design.

Some object-oriented design methods are supposed to be applied to the results of an object-oriented analysis. Others exist which assume a preliminary functional analysis. Because of this, object-oriented design methods have different starting points.

Object-oriented analysis is concerned with understanding the user requirements and modelling them by identifying problem-space objects. Most OOA methods suppose the existence of a requirements document [CY91a, RBP+91] while others may start analysis by writing the user requirements [Ber89, RG92]. Høydalsvik and Sindre argue that an OOA method which starts based on an existing requirements document is largely a high level design method [HS93].

If the design method is based on the results of an object-oriented analysis, then the analysis method will already have identified and described the problem-space objects. The design method takes those objects and decides which are to remain in the final system; adds new objects which only belong to the solution domain and whose purpose is to support the implementation of the objects of the problem-space; and finally decides about the implementation of each object. In this case, the transition between analysis and design is not a problem.

If the design method is based on the results of a functional analysis, then it must identify the problem-space objects before it can deal with the solution-space. Therefore, the object-oriented design method must perform a task which is really the concern of the analysis phase.

As an example, Booch's Object-Oriented Design may suppose a functional analysis document as a starting point while Object-Oriented Design by Rumbaugh *et al.* [RBP+91], for example, supposes the existence of an object-oriented analysis document. The objects identified by Booch's method are problem-space and solution-space objects whereas the objects identified by OMT are solution-space objects, as the problem-space objects were identified in the analysis phase (see [MC92] for further information).

In both cases, it is the design phase which is responsible for the identification of solution-space objects and for the decision of whether or not each problem-space object is to be kept in the solution.

Looking at functional methods for analysis and design we can identify an interesting concept, which seems to have been forgotten in object-oriented methods. Functional methods usually talk about the ideal system and the real system (sometimes called the logical and the physical system) [DeM79, GS79]. The ideal (logical) system is proposed during the analysis phase and it reflects the system without any constraint, i.e. in an ideal environment. In an ideal environment, we could add that nothing unexpected can happen

to the system. This could be seen as a good reason to deal with erroneous situations in a later stage of the development.

The real (physical) system is proposed during the design phase and it should reflect the environmental constraints in which the system is going to operate. However, this can be dealt with in the last steps of functional analysis methods if they are proposed without any design method. Therefore, in general, analysis and design can overlap if their methods do not take into account the design method that follows analysis, or the analysis method that precedes design.

This idea of "ideal" and "real" is interesting, and could be brought into object-oriented methods in order to help clarify the distinction between the phases. Although the idea of "problem-space" and "solution-space" does correspond, to some extent, to the "ideal" and "real" concepts, it is worth noticing that the problem-space objects are usually based on requirements documents which incorporate constraints.

2.2.4 The Origins of Object-Oriented Methods

A significant question is whether or not existing object-oriented analysis and design methods are implementation language dependent. One thing is what they *should be*, the other is what they *really are*.

It is generally agreed that analysis methods should not be implementation language dependent, while design methods can be. However, this does not always occur in practice. The reason is easy to explain. The evolution of the software development life-cycle was backwards, i.e. from programming to design to requirements analysis. It is no different now; object-oriented development has its roots in object-oriented programming. Many of the concepts in the analysis and design methods therefore come from programming languages. If the programming language emphasises certain concepts, then authors will try to include them in the analysis (or design) methods. This is not a new situation, many functional methods were developed with the eventual implementation very much in mind.

We group object-oriented methods into two classes, depending upon their authors (and users): Ada community methods and object-oriented programming language community methods. Object-oriented analysis methods such as the ones from Berard [Ber93] and Colbert [Col89] expect the eventual implementation to be in a language like Ada, and inheritance plays a weak role in their method. We used the method Object-Oriented Requirements Analysis from Berard in our previous work to develop an object-based Message Switching System in Ada [FMG89, FMG90a, FMG90b, GFM89, MFG89, MGF90]. On the other hand, the object-oriented analysis method from Coad and Yourdon is oriented towards an implementation in a programming language which supports inheritance. Champeaux and Olthoff go further and state that object-oriented analysis "acknowledges that an implementation is done in an object-oriented programming language" which would seem to deny the advantages of the approach to the Ada community [dCO89].

2.3 Formal Methods and Executable Specifications

2.3.1 Classification of Approaches

Specifications can be classified according to three criteria:

- Functional versus Object-Oriented (or Object-Based).

- Formal versus Informal.
- Executable versus Non-Executable.

Representatives from each pair can be combined in any order. For example, functional methods can lead to informal non-executable functional specifications. Formal methods such as Object Z [CDD +89] lead to formal non-executable object-oriented specifications. We are interested in formal, executable, object-oriented specifications [CM94, MC92, MC93a, MC93b, MC93c, MC94a, MC94b, MC94c, MC94d, MLC94].

Functional versus Object-Oriented

Functional and object-oriented methods can produce approaches which we call:

1. Functional when we use functional analysis and functional design.
2. Hybrid when we use functional analysis and object-oriented design.
3. Object-oriented when we use object-oriented analysis and object-oriented design.

The functional approach has been applied for the last twenty years. During this time, many methods for both the analysis and the design phases have been proposed and CASE tools created to support these methods. The most widely used functional analysis methods are Structured Analysis by de Marco [DeM79], Yourdon [You89] and by Gane and Sarson [GS79]. Although many people have acquired a good understanding of this subject, some problems still remain. The main problem is that functional methods do not provide a single representation for the processes and for the data.

The hybrid approach was commonly used in the late eighties, when object-oriented design methods were applied together with functional methods. This creates problems which sometimes can be difficult to solve. With a large system, the requirements specification will be divided into several components each of which will be dealt with by a separate team member. If object-oriented design is being used in the design phase, and each of the components identified in the (functional) analysis phase is given to a different member of the design team, there can be major integration problems when the system is finally put together.

As the concepts used by functional methods and the concepts used by object-oriented methods are so different, the transition from a functional analysis to an object-oriented design is very difficult.

Although many object-oriented analysis (and design) methods have appeared in the last few years, people are not keen to change completely from a functional culture to an object-oriented culture. One of the reasons is that there is not proof that the object-oriented methods are better than the functional methods, for example. It is recognised that object-oriented methods are not well suited to describe all kinds of problems. Also, it is necessary to invest more to develop CASE tools at least as good as the existing ones supporting functional methods. For these reasons, object-oriented analysis and design methods are being introduced gradually but slowly in industry [Sta93, WRW93].

Because the concepts used in object-oriented analysis and design are the same, the gap between the analysis phase and the design phase is very narrow [Ber93]. Moreover, the techniques used by the object-oriented design methods usually produce designs which

are very close to program text. Sometimes they already are outline code, as when Ada or Eiffel is used as a design language. However, it is more difficult to separate "analysis concerns" from "design concerns", as the concepts and techniques used have much more in common than is the case with functional methods [Ber93, p47].

Formal versus Informal

The primary benefit of formal techniques is that, as they have a precise mathematical semantics, the resulting specifications are unambiguous. In contrast, informal techniques lead to specifications which leave much of their interpretation to the reader. The imprecision of an informal specification can give the implementor a freedom of interpretation which can cause errors and omissions in the code, resulting in high costs for support and repair. (According to Boehm, correcting one error in an early stage of the development is much cheaper than correcting the same error in the code [Boe87, Boe81].) Moreover, this imprecision can lead to misunderstandings in validating the informal specification against the requirements. A formal approach to specification is therefore useful, in that it allows design and eventual implementation to be verified against the specification, at least in theory, although it still leaves the problem of validating the specification against the initial informal requirements document.

Proving that a requirements specification for a large and complex system, a design specification and the eventual implementation all describe exactly the same system is beyond the current state of the art. A practical approach is to make the specification executable and perform the validation by means of conformance testing where a series of interface scenarios are used to show that the different specifications and the final implementation all exhibit the same behaviour.

Executable versus Non-Executable

We believe that if a requirements analysis specification only says *what* the system does, then it cannot be executed. In order to have an executable specification we must introduce some *how* in it. Some software engineers propose that specifications should not be executable, because a specification written in a notation that is directly executable will contain more implementation detail than a non-executable one [HJ89]. There is also the danger that executable specifications can overspecify a problem. There are two reasons for this. Firstly, an implementor may be tempted to follow the algorithmic structure of the specification and, secondly, the executable specification may produce particular results in cases where a more abstract specification might allow a number of different results.

There are other authors who believe that the use of formal and executable specifications only bring advantages, and no disadvantages, when developing software [Fuc92, Zav84, Zav91, ZY81]. Also, being able to demonstrate that a specification exhibits the expected behaviour can greatly increase one's confidence in it [Fuc92]. The accusation that this is no more than testing is partially answered by using symbolic evaluation. The LOTOS SMILE simulator [EW93] supports symbolic execution.

Some formal description languages can be executed, others cannot. There are some whose main feature is that they are executable, e.g. *me too* [AJ90], there are others of which a large subset is executable, e.g. LOTOS (Language of Temporal Ordering Specification) [ISO88], while others cannot be executed, e.g. VDM (Vienna Development Method)

[Jon86]. Informal functional analysis techniques lead to requirements specifications that are not executable.

If a specification is executable, then it can act as a prototype of the system. One can raise the question about whether or not a prototype must be executable on a computer. For some authors prototyping can be done manually [Gib90], for others it must be automatic [AJ90]. In Section 2.3.4 we present a further discussion on prototyping and executable specifications.

2.3.2 The Benefits of Combining OOA and Formal Methods

The importance of formal methods has increased in the past few years. A recent study about the use of formal methods in twelve industrial systems concluded that [CGR93]:

> [...] formal methods, while still immature in certain important respects, are beginning to be used seriously and successfully by industry to design and develop computer systems.

Nevertheless, the use of formal methods is not yet satisfactory. Another survey based on the available literature and on enquiries in industry took place to find out the reasons for the weak acceptance of formal methods [AP93]. The results showed that there are major benefits of using formal methods and a few significant limitations. Among the benefits of using formal methods, we would like to emphasise that formal methods:

1. produce specifications which are unambiguous;
2. help to discover inconsistencies, ambiguities, omissions earlier in a project life-cycle;
3. can, in principle, be used to verify that a program is correct, according to the properties shown by a specification;
4. can be used to build an implementation by successive mathematical transformations of a specification.

The major limitations for using formal methods seem to be the need for mathematicians to do proofs and perhaps the lack of tools. However, the specification languages are not much more difficult to learn than programming languages and the proofs are not always necessary [AP93, Hal90].

Most currently used software engineering practices lack formality. We believe that combining formal methods with the object-oriented paradigm can be a fruitful approach when modelling and developing large software systems. On the one hand, by formalising the object-oriented models, formal methods force us to be rigorous about the meaning of each system component and not leave modelling decisions to be made at the implementation stage. On the other hand, by using the practical OOA methods with formal methods we are making the formal specification languages more acceptable for use by a larger community.

Craigen *et al.* support this view by recommending, among others, the following directions of research [CGR93]: (a) improve the integration of formal methods with other software engineering practices, such as object-oriented programming; (b) develop notations more suitable for use by individuals not expert in formal methods or mathematical logic.

ROOA proposes a mapping of the object-oriented analysis constructs into LOTOS which helps users in the use of the formal specification language.

Another area of research, this time suggested by Plat *et al.*, is the construction of an initial formal specification in the preliminary design [PKT92]. According to these authors, none of the known formal methods provide guidelines about how a requirements specification can be constructed. In fact, the use of formal methods is often delayed until the design phase with the result of the analysis phase being an *informal* requirements specification. In ROOA we combine OOA methods with LOTOS to produce an initial *formal* requirements specification during the analysis phase. As the resulting specification is executable, the difficult task of validating the informal requirements against the specification is simplified. Executable specifications also result in a greater involvement by the users in a software project [Fuc92].

Work has been done on how to get a first formal specification. An example is the SPECS project, whose goal was to support the formalisation of requirements by creating a specification generation process [Gen92]. However, the starting point of this specification generation process is an informal requirements specification, which therefore requires a previous analysis phase. The resulting method is called the CR&F method.

2.3.3 Reasons for Choosing LOTOS

As our goal is to integrate formal description techniques with object-oriented analysis methods so that they can be used by software engineers, we believe that the chosen formal description technique should satisfy the following conditions:

- it is an ISO standard;
- it is able to produce specifications in an object-oriented style;
- support tools are available;
- the specifications are executable so that prototyping can be used;
- it supports concurrency.

The specification language LOTOS satisfies these conditions.

As we said before, a major advantage of the object-oriented approach is that it supports the direct modelling of real world entities as a set of autonomous objects which communicate with one another by sending messages. Therefore, an object-oriented analysis model should represent the requirements as a set of communicating concurrent objects even when the eventual implementation is to be sequential. The formal language used to represent the formal model should support this view.

LOTOS has a process part and a data typing part. The process part is based on a combination of the language CSP [Hoa85] and of the language CCS [Mil89]. The data-typing part is based on ACT ONE [EM85]. Processes communicate by synchronising on events during which information may be exchanged. As there is a straightforward mapping between concurrent objects and process instances and between message passing and event synchronisation, LOTOS is well suited to representing the requirements as a set of communicating concurrent objects. Inheritance is more difficult to represent, but can be modelled as well. We believe that a practical method should be based on standard

languages, so we have not used any of the suggested object-oriented extensions to LOTOS (e.g. [Rud92]).

Prototyping is a useful tool in validating a specification against a set of requirements. LOTOS offers an extensive set of tools, such as syntax checkers, semantic checkers and simulators. The SMILE simulator [EW93] supports non-determinism and value generation. This allows symbolic execution of a specification where a set of possible values is used rather than particular values. Many more behaviours can then be examined with each simulation than is possible when all data values have to be instantiated. SMILE uses a narrowing algorithm to determine when a combination of conditions can never be true. As we are in the analysis phase, we can use non-determinism to model behaviour so that premature design decisions are not made.

During the last few years, several object-oriented specification languages have been proposed, such as Object Z [CDD $^{+}$89] and TROLL [HG94, JSHS91]. However, none of the currently existing languages, for one or other reason, satisfied the proposed criteria.

2.3.4 Prototyping

Informally, the term *prototyping* can refer to a tool (e.g. SMILE [EW93]), a specification language (e.g. LOTOS [ISO88]), an approach (e.g. spiral model [Boe88]) or the actual building process. The end product is the *prototype*.

In engineering, 'prototype' can mean:

- a model of the problem in a reduced scale (for example a dam or a bridge); or
- a product built (by hand) before production on a large scale.

Neither of these meanings can be directly applied to software. According to the IEEE Standard Glossary of Software Engineering Terminology, a prototype is a preliminary type, form, or instance of a system that serves as a model for later stages or for the final, complete version of the system [IEE91]. When speaking about the prototype of a software system, we use this expression to denote a model of the system to be implemented. Hence, a prototype should be executable [AJ90, BGW82, Cho92, JDP89, TY92]. The term *rapid prototyping* is often used to indicate the early execution of the specification. A (rapid) prototype differs from the actual implementation in that it focuses on the functionality of the problem rather than on efficiency.

In software engineering we can imagine prototyping to be useful for:

- exploration;
- experimentation;
- evolutionary development;
- throw-away development;
- a pilot project.

The borderline between *exploration* and *experimentation* is fuzzy. Exploration is used to clarify requirements and desirable features of the target system, while experimentation is used to determine whether or not a solution is adequate before we start developing the large

scale implementation [Flo84]. Both cases are used for validation of ideas. The resulting product obtained by experimentation may be thrown away if the environment where the prototype was produced is not integrated with that of the final product. The prototype obtained by exploration is normally thrown away, as it is messy and unstructured.

The *evolutionary* approach (see [Dav92, IH87, Smi91]) produces a sequence of versions in which each version is a refined version of the previous one. The prototype is built in a quality manner (including software requirements specification, design documentation, and thorough test) [Dav92]. The well-understood parts are built first and the parts not understood well are left for further iterations. There are some authors who differentiate between evolutionary and *incremental* development [Flo84, Gra89, IH87]. The incremental development deals with a problem by stepwise extension. The main difference between these two approaches is that with the evolutionary approach the design is allowed to evolve throughout the use of the system, whilst with the incremental approach the design is frozen and the only changes accepted are those due to implementation errors. This means that an evolutionary approach also contemplates the maintenance phase and so any changes introduced during this phase are dealt with as if the system was still in its early days of development. However, the incremental approach does not deal with changes introduced during the maintenance phase. We do not distinguish between the evolutionary approach and the incremental approach.

In the *throw-away development* proposed by Brooks, the functionality of a system is simplified in order to decrease the time of prototyping [Bro75]. The parts of the system not well-understood are built quickly, not paying too much attention to efficiency. However, by doing this, the parts of the problem which can make the system fail may be eliminated. The temptation to give the prototype to the client, by pressure either from the client or from the managers, should be resisted. In his classic "The Mythical Man-Month", Brooks advises *plan to throw one away; you will, anyhow* [Bro75, p116] [1].

Finally, prototyping can be used to build a *pilot project*. A pilot project can be used to support the introduction of new technology. It accepts higher risks during development, such as delays and extra costs, and the final product is delivered to the client.

In summary, we believe that the throw-way and the evolutionary approaches are the two primary schools of prototyping. In the throw-away approach the prototype software is constructed in order to learn more about the problem and its solution. The resulting working model only shows some of the features of the final system. The idea is to determine characteristics of the system, to estimate costs, to establish feasibility and performance limits and, if necessary, to explore different designs and interfaces. After the desired knowledge has been obtained, the final product is discarded. Followers of the throw-away approach are [BGS84, Dav82, Gom83, GS81].

In the evolutionary approach, the prototype is also constructed in order to learn more about the problem and its solution. However, once the knowledge has been gained, the prototype is adapted to satisfy the now better-understood requirements. Then, the prototype is used again, more is learned, and, once more, the prototype is re-adapted. This process is repeated indefinitely until the prototype system satisfies all needs. This approach seems to respond to the inevitable change of requirements during the development and operation of the system. Followers of the evolutionary approach are [BGW82, BT75,

[1] Brook's law of prototypes: *plan to throw away*; In [Ben88, p64], Zerouni says: *if you plan to throw away one, you will throw away two.*

MC83, TY92, Zel80].

Prototyping with Formal Executable Specifications

The above description of prototyping is very general. It does not focus on any special phase of the software life cycle or on any prototyping tools. We can use prototyping at any stage of the software development, if we have the appropriate tools [Rat88]. In this thesis we are interested in executable specifications to be used as prototypes during the analysis phase.

We are using prototyping to help:

- verify the internal consistency of a formal specification;
- validate the specification against the requirements.

Boehm says that, when classifying the top 10 list of software metrics in terms of their value in industry situations [Boe87, p84]:

> Finding and fixing a software problem after delivery is 100 times more expensive than finding and fixing it during the requirements and early design phases.
>
> This insight has been a major driver in focusing industrial software practice on thorough requirements analysis and design, on early verification and validation, and on up-front prototyping and simulation to avoid costly downstream fixes.

He considers this to be the factor number one in his top 10 list.

ROOA allows us to use prototyping from the very early stages of the development. It uses an evolutionary approach to incrementally produce a formal requirements specification expressed in LOTOS. As the resulting specification is executable, we use the LOTOS tools as prototyping tools. The approach we follow can be seen as a horizontal development where each specification obtained is refined and validated to produce a more complete one (see Figure 2.2).

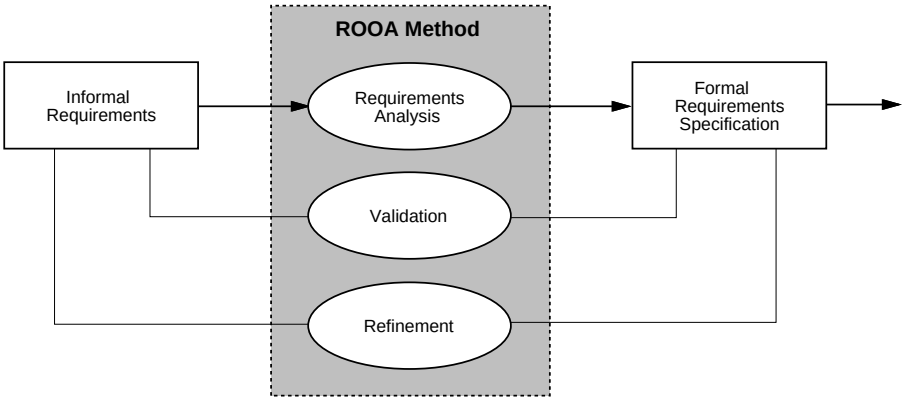

Figure 2.2: ROOA's horizontal development

We are concerned with the production of a specification within the requirements life cycle. The requirements life cycle can be seen as comprising three main development

stages within the analysis phase [NW93]: *problem identification*, *modelling activity* and *analysis activity*. Dubois *et al.* [DDBP93] divide the analysis activity into *analysis* (to test whether the conceptual model satisfies formally required quality criteria) and *validation* (to test whether the conceptual model meets the informal user requirements). The problem identification includes the understanding of the informal requirements, if they already exist, and, if they do not exist, the capture of the requirements by studying any available documents which describe parts of the problem and by discussions with the customer. The ROOA method accomplishes the modelling activity by using object models, event trace diagrams, object communication tables, object communication diagrams and LOTOS to produce an initial formal model of the required system. ROOA is then concerned with the analysis activity, by refining and validating the requirements specification built during the previous stage. To accomplish this, we use prototyping.

From the Formal Specification to the Implementation

After having the formal requirements specification produced by ROOA we can follow two different approaches to the implementation. First, we can follow the Swartout [SB82] approach by taking the requirements specification, creating from there a design specification and then an implementation, allowing feedback in any stage.

Secondly, we can use the requirements specification as the first stage in a formal development, by following a transformation trajectory from specification through design to implementation. Each transformation starts with a *complete* formal specification and adds to it design (and later implementation) detail and creates a new version of the formal specification which preserves the behaviour of the previous one.

At each stage we can use prototyping to ensure that each specification conforms to the previous specification and to verify internal consistency of the specifications. This approach has its roots in the work proposed by Balzer *et al.* [BCG83]. It is also the VDM approach where proof obligations, and not prototyping, are used to show that the specifications conform to the higher level ones.

ROOA is concerned with producing the initial formal specification and not with the resulting development trajectory.

2.4 Conclusions

This chapter introduced the background for the thesis and created the context for the work we present in the following chapters. It can be divided into two main parts. The first part started by analysing the software development concept, by giving an overview of the software life-cycle and software development models, and then focused on the differences between analysis and design. Then, it concentrated on object-oriented approaches and related functional analysis (structured analysis) with object-oriented design, highlighting the need for an integrated object-oriented framework.

The second part discussed formal methods and executable specifications. It began by classifying the existing approaches according to the kind of methods used and the resulting specifications produced. It followed by examining the benefits of combining object-oriented analysis with formal methods. Next, it showed the reasons to choose LOTOS as the ROOA specification language. Then, it presented several views of prototyping and indicated the

advantages of having formal and executable specifications. Finally, it introduced two ways to go from a formal requirements specification to an implementation.

Chapter 3

Modelling Fundamental OOA Concepts in LOTOS

3.1 Introduction

LOTOS does not incorporate object-oriented constructs such as inheritance and dynamic binding. It was not one of the goals of its designers, since LOTOS was designed before object technology became widely established, although object-oriented programming has been around at least since Simula, in 1967. Nevertheless, the language directly supports the concepts of encapsulation, abstraction and information hiding which provide a basis for writing specifications in an object-oriented style [Bla89, CJ92, CRS90, May89, Rud92]. In LOTOS, concurrent objects may be modelled as process instances, composed by using the parallel operators, and message passing may be modelled as two processes synchronising on an event.

This chapter studies how the following object-oriented analysis concepts can be mapped into LOTOS:

- Class templates, classes, abstract class templates and objects;
- Attributes, object identity and state;
- Services, methods and behaviour;
- Environment of an object and external objects;
- Events and message connections (communication);
- Inheritance;
- Conceptual relationships;
- Composition and decomposition;
- Aggregates;
- Subsystems.

In Section 3.2 we introduce each of these object-oriented concepts and in Section 3.3 we give an interpretation of them in LOTOS, leaving to Chapter 4 a detailed study of aggregates. The basic concepts of class template, service, attribute, class and object are then described formally in Chapter 5.

3.2 Object-Oriented Definitions

As there are many different definitions of object-oriented concepts in the literature, we give here a coherent set of definitions which are used in the rest of the thesis. These definitions are based on the Open Distributed Processing Reference Model and on a general understanding of object-oriented analysis concepts [ISO94, Rud93].

3.2.1 Class Templates

A class template describes the common static and dynamic properties of objects of the same kind (belonging to the same class).

Each node in the object model proposed by most of the OOA methods describes a class template, as opposed to an object or a class.

3.2.2 Classes

A *class* is the set of all objects which share the common features specified by a class template. This definition also includes the notion of *subclass*, since the set can be a subset of all objects which possess the common features specified by a (*super*-)class template.

3.2.3 Abstract Class Templates

An abstract class template is a (super)class template which has no objects, i.e. it cannot be instantiated. An abstract class template is used in the definition of subclasses. The restriction of having no instances does not apply to the subclasses.

Some authors propose a special abstract class (template) symbol to be used in the object model [CY91a].

3.2.4 Objects

An object is a member of a class and is created by instantiating the class template.

An *object* can be used to model either a real world entity or a concept. It combines structure with behaviour in a single encapsulated entity which can be characterised by *name*, *state information* and *services* (or *operations*) ([CY91a, p53], [Jac92, p44], [RBP +91, pp22–26]). The name gives *identity* to the object and is used to reference it. The state information is given by the values of the data types (attributes) encapsulated by the object. The services constitute the interface of the object.

An object has an internal and an external view. During the analysis phase we are mainly concerned with the object's external view and with describing the behaviour of each object in an abstract way. During the design and implementation phases we are mainly concerned with its internal view.

The external view of an object corresponds to its interface and includes only those properties which the client objects need to see. The internal view of an object corresponds

to its implementation and it reveals the underlying structure of any stored data, the details of the algorithms used to accomplish the services, and the underlying layers of abstraction used to implement it. The designer of the object knows both the internal and the external views. The users of an object only know its external view.

3.2.5 Attributes

An attribute defines a static property of a class template, describing a data value held by each object of the class. The attribute values give the object's state information. Each attribute name is unique in a class, but attributes in different classes can have the same name.

3.2.6 Object Identity

Each object is distinct from any other object. Two objects can have the same attribute values and offer the same services, but they will still be different from each other. Rumbaugh *et al.* define identity as [RBP +91]:

> [...] a distinguishing characteristic of an object that denotes a separate existence of the object even though the object may have the same data values as another object.

This distinguishing characteristic is referred to as the *object identifier*.

3.2.7 State (of an Object)

The state of an object, at a given time, is determined by the values of its attributes ([CY91a, p145], [Jac92, pp228–229], [RBP +91, p84,p87]) together with the conditions that determine the services which the object currently offers. The values of the attributes are related to the static aspect of the object, whilst the conditions control the possible state transitions and therefore are related to the dynamic aspect of the object.

3.2.8 External Objects

An external object is an object which does not belong to the system being analysed, that is, it is an object outside the problem domain. The set of external objects provides the environment for the entire system.

3.2.9 Environment (of an Object)

The environment of an object is formed by all the other objects which constitute the system; i.e. it is the part of the model which is not part of that specific object. An external object is not part of the environment of any object in the system.

3.2.10 Services

We distinguish between offered services and required services. An offered service is a capability that an object exports and which can be used (called) by other objects in the model or by the actors (or external objects) of the system. A required service is a service

that an object requires from another object. This service is defined in the second object as an offered service. The services are the only mechanism other objects (inside or outside the system) can use to change or to query an object's state. Therefore, an object interacts (communicates) with other objects via services.

It is common in the object-oriented community to classify the services offered by an object into three categories: *constructors*, *modifiers*, and *selectors*. A constructor creates an object, and usually initialises its state. A modifier has the ability to change the state of the object in which it is encapsulated. A selector returns state information about the object, but does not change the state.

3.2.11 Methods

We differentiate between *services* and *methods*. A service is contained in the interface of an object (or class template) and advertises an object's capability. A method is internal to an object (or class template) and is the actual mechanism by which the service is accomplished.

3.2.12 Behaviour (of an Object)

The behaviour of an object describes the dynamic conduct of the object during its life time. It is described in terms of the interactions the object can have with other objects, the order in which these interactions may occur and the way the state information of the object changes. Conditions that determine the events in which the object can take part restrict that object's behaviour.

3.2.13 Events

An event is something that happens instantaneously at a point in time. An event has no duration, compared with the time granularity in which we are interested.

In general, two events may, or may not, be related by precedence. That is, one event may precede or follow another, two events may occur simultaneously, or they may be completely independent.

3.2.14 Message Connections

Objects interact (communicate) with each other by sending messages. Some OOA methods use the term *message connection* to mean communication between two objects. A message connection reflects a dynamic processing dependency between an object and the other parts of the system. Message connections are represented in an object model as arrows. If a message connection is defined from object A to object B, it means that object A requires services from B to accomplish its behaviour and the arrow points to B.

The mechanism objects actually use to communicate between each other depends upon the language used. In a sequential language, such as Smalltalk, objects communicate via message passing while in concurrent languages, such as Ada, communication is achieved by *entry calls*.

3.2.15 Inheritance

Generalization/specialization, *is-a*, and *subtyping/supertyping* are some of the terms used to denote an inheritance-like concept.

There are two main definitions of inheritance: *behavioural inheritance* and *incremental inheritance* [ISO94]. Behavioural inheritance is related to the typing concept. Incremental inheritance is concerned with the the creation of a derived class template by the modification of a parent class template.

We say a *superclass* is defined by the existing class template and a *subclass* is defined by the newly defined class template which inherits all the services and attributes of the superclass and, in addition, can redefine inherited services and add new attributes and new services to the inherited ones.

Incremental inheritance provides the capability to allow objects to be specialised from existing ones. It is based on the idea of incrementally modifying existing class implementations. It mainly supports the concept of reusability and software engineers use it to define new classes from existing ones, even when no subtyping relationship is intended. Incremental inheritance is therefore used as a mechanism to share code and the services offered by the newly defined class. In LOTOS, behavioural inheritance is achieved by restricting incremental inheritance [Cla94c, CM94, MC93a].

In behavioural inheritance, objects of a subclass inherit all the services and attributes defined in their superclass and can be used wherever an object of the superclass is expected. Objects of a subclass can extend the inherited properties by defining new services. Redefinition of services is also allowed, but only if the signature and the semantics of the redefined service conform to the signature and semantics of the service in the superclass. A signature conforms to another if the number and the type of the parameters of those services are the same, and also if the returned results, if any, are of the same type. For the semantics to conform, the redefinition of services must also obey two extra conditions [BD92]:

1. The redefined service must return the same value as the original service in the superclass when applied to the base part of the subclass object. (The base part of the subclass is the part of the subclass, attributes and services, which is also defined in the superclass.)

2. Let b be a superclass object with the same initial state as the base part of a subclass object d. Redefinition must change the state of the base part of d in the same way as the superclass service changes the state of b.

As we are in the analysis phase, we are concerned with the sharing of properties, not the sharing of code, even if we can use incremental inheritance to implement behavioural inheritance. Some authors have argued that we should have separate behavioural inheritance and incremental inheritance hierarchies, as their similarity is a source of confusion [Ame91, DT88, Ier93, Por92].

The discussion above can be extended to include the ability of a subclass to inherit properties from several superclasses. This is called *multiple inheritance*.

3.2.16 Conceptual Relationships

There is an open-ended requirement for conceptual relationships (or associations or static relationships). They are the most common type of relationships and are application de-

pendent, describing the role that one object plays with respect to another. For example, in a banking problem, we could define the relationship *has a* between the object *account* and the object *card*, and read it *account has a card*. These relationships are characterised by their names and their cardinality. The cardinality is defined in terms of an upper bound which can be $1:1$ (to be read one-to-one), $1:N$ (one-to-many), and $N:M$ (many-to-many) and a lower bound which can be $0:0$ (zero-to-zero), $0:1$ (zero-to-one) and $1:1$ (one-to-one). The lower bound specifies the minimum number of mappings between objects. It can be studied by taking into consideration the two objects involved in the relationship and called for example $0:1$, or it can be studied concentrating only on one of the objects involved. In this case, authors use the terminology optional (if it is a "zero-side") or mandatory (if it is a "one-side").

We combine the lower bound and the upper bound, separated by a comma, and we use this combination in the object model beside the class template of the objects to which it refers. Figure 3.1 shows two class templates with objects, using Coad and Yourdon's notation, connected by a line that indicates a relationship. The numbers drawn in both extremes of this line represent the cardinality of the relationship. The maximum cardinality (upper bound) of this relationship is $1:M$ and the minimum (lower bound) is $0:1$, reading from the left to the right [1].

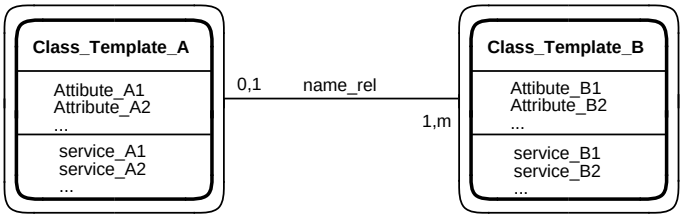

Figure 3.1: A relationship between objects

This is supposing that the relationship is defined implicitly in both directions, i.e. bidirectional. While some methods give preference [RBP +91] to bidirectional relationships, there are others [Jac92] which suggest that it is better to define each relationship in a single direction, i.e. unidirectional. ROOA accepts object models built by using any of the existing object-oriented analysis methods, and therefore it can deal with both bidirectional and unidirectional relationships. As we will see in Section 3.3.11, the exercise of modelling one or the other kind of relationship in LOTOS is similar.

An association between two objects, where each one belongs to a different class (a binary association), means that one object knows about the other or, if it is bidirectional, they know about each other. We can have a unary association, if the two objects involved in the relationship belong to the same class, and a complex association, if more than two objects (from different classes) are related. Complex associations can always be transformed into a collection of binary associations by creating another object to which each of the existing objects would relate.

Different associations between the same objects can coexist in the same model. In this case we should keep them separated and model them independently in order to give the right semantics to the model.

An association gives a potential for communication to the objects involved.

[1] If the maximum and the minimum on the same side are both *one*, we write '1', instead of '1, 1'.

Binary Relationships

A binary relationship is defined between two objects, each one belonging to a different class. Let us consider the maximum cardinality. In mathematical terms, a binary relationship R between the two sets S_1 and S_2 is called $1:1$ when each element of S_1 is related to at most one element of S_2 and the same condition holds for the inverse relation R^{-1}; it is called $1:N$ when each element of S_1 can be related to many elements of S_2 and each element of S_2 is related to at most one element of S_1 by the inverse relation; and any relation can be called $M:N$, i.e. there is no restriction and therefore each element of S_1 can be related to many elements of S_2, and in the R^{-1} relation, each element of S_2 can be related to many elements of S_1. Therefore, every $1:1$ relation is a $1:N$ relation and every $1:N$ relation is a $M:N$ relation. What is important is to determine the most constrained group to which the relation belongs.

This study can be extended to include the lower bound.

Unary Relationships

Unary relationships involve two objects of the same class [2]. This type of relationship can be sometimes seen as an inheritance relationship.

Note that, unary relationships give us the potential to define communication between two objects of the same class. This is important, since message connections used by the most common object models only define communication between objects of different classes [CY91a, Jac92].

Relationships with Values

There are situations in which a specific piece of information does not belong to any of the objects involved in the relation, but it only exists because those objects are related; i.e. it belongs to the relationship itself. In such a situation we say that a relationship holds values. This relationship may be binary or unary.

3.2.17 Composition and Decomposition

An object-oriented system can be regarded as a collection of interacting objects each of which is a member of a class. The concepts of class, inheritance and aggregation (see Section 3.2.18) are very helpful in managing software complexity, but they are not enough. If we are constructing a large and complex system then, in order to control complexity, it is necessary to group objects into subsystems. Whether this is frequent depends upon the style of the original requirements document. The description of the system may be mainly "flat", i.e. without hierarchical structure, in which case grouping the objects into subsystems is important in understanding and structuring the overall system [Mil56, Par72, You82].

There are two particularly important structuring techniques that we can apply to collections of objects:

- Composition, which combines objects to form larger objects;

[2] One could expect that a unary relationship would involve only *one* object. This is not the case. In fact, what gives the name to the relationships is the number of classes included.

- Decomposition, which refines larger objects into component objects.

A bottom-up development makes heavy use of composition and a top-down development makes heavy use of decomposition. Which technique is more useful for a particular development depends on the size and complexity of the problem, but also, in a first iteration, on the style of the requirements. We can use a mixture of both. We can first use decomposition to divide the system into subsystems and later use composition to build subsystems or composite objects from simpler (perhaps already existing) objects; for example, objects reused from another context.

3.2.18 Aggregates

Aggregation is a special form of relationship, and not an independent concept. It is a *part_of* relationship in which the *aggregate* (composite object) is made of parts (object components).

We can distinguish two situations when dealing with aggregation: (i) the object components are only known by the aggregate and therefore no associations or message connections are defined between the other objects of the system and the components; (ii) the object components are shared by other objects in the system, having associations and message connections with those objects.

In the first case, where the object components are not shared, the object components are *hidden* from the rest of the system and they belong exclusively to the aggregate. The interface with the rest of the system is made via the composite object.

In the second case, where the object components are shared, the aggregation relationship should be seen as a regular conceptual relationship, where each object component is related to the composite object.

In both cases, the aggregate and each object component have their own identity.

One of the goals we had in mind when developing ROOA was to produce a method which could be applied to model large systems. For this reason, aggregates play an important role in ROOA, as they are the main mechanism to model complex objects. Therefore, the decision we made of dedicating a whole chapter, Chapter 4, to deal with aggregates and to show how we model them in LOTOS is not a surprising one.

3.2.19 Subsystems

A subsystem is merely a grouping of class templates and is only created to structure the system, helping us manage complexity. The difference between an aggregate and a subsystem is that an aggregate is an object (and is defined by a class template) while a subsystem is not and therefore it has no identifier.

Subsystems are like transparent boxes which are not visible to the rest of the objects in a system. Therefore, other objects communicate directly with the object components of a subsystem as if it did not exist. Subsystems only exist to guide the reader's attention through the system.

Class templates are grouped to form subsystems by following the concepts of *coupling* and *cohesion*. Coupling measures the "strength of interconnection" among subsystems and cohesion measures how tightly bound or related the components of a subsystem are to one another. Ideally, we want loosely coupled subsystems so that we may treat each

subsystem relatively independently of the others, and strongly cohesive subsystems so that the components of a given subsystem are functionally and logically dependent.

3.3 Mapping Object-Oriented Concepts into LOTOS

As LOTOS contains two distinguishable parts, processes and abstract data types, a choice must be made as to which part is better suited to model an object, a class template and a class. In ROOA, a class template is specified either as:

- A process and one or more ADTs: the process describes dynamic behaviour and the ADTs, given as parameters of the process, describe state information.
- A single ADT: when an object only plays the role of attribute of another object, it is modelled simply as an ADT.

Objects which have an important role in the system are always able to communicate with other objects and may receive communication from other objects. The class template that defines this kind of object is defined with a process and one or more ADTs.

Objects in the object model which do not need to communicate with other objects are considered by ROOA as attributes; and, in ROOA, an attribute is modelled by an ADT. These 'objects' from the object model are then values of ADTs. These values are kept as parameters of processes which model objects with an important communication role. (In fact, these ADTs may be used to specify more complex ADTs which then are used as parameters of some processes.)

In the following sections we present the LOTOS interpretation of the object-oriented concepts described in Section 3.2 by means of a running example.

3.3.1 Automated Banking System Example

The example we have chosen is an automated banking system. A brief outline of the problem is given here.

> Clients may take money from their accounts, deposit money or ask for their current balance. All these operations are accomplished using either automatic teller machines or counter tellers. Transactions on an account may be done by cheque, standing order, or using the teller machine and card. There are two kinds of accounts: savings accounts and cheque accounts. Savings accounts give interest and cannot be accessed by the automatic tellers.

This problem has been first analysed using the method proposed by Coad and Yourdon [CY91a]. The final object model is depicted in Figure 3.2.

3.3.2 Class Template

A class template (or template for short) embodies the common characteristics of objects of the same kind. It specifies what constitutes a typical object, without individual identity.

A template is defined in LOTOS by specifying a process definition. The process definition may have formal parameters which are part of the common features of that kind of

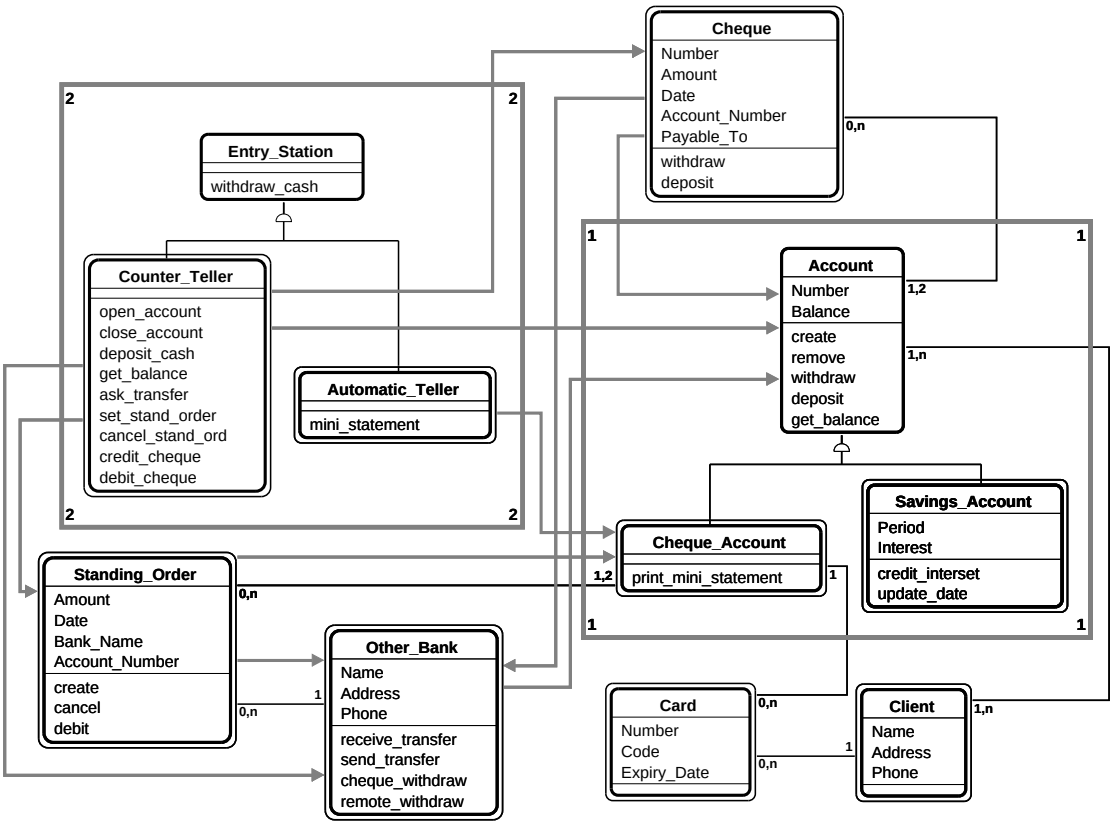

Figure 3.2: A Coad and Yourdon object model for the banking system

object. These parameters, which are ADTs, define the state of the object. In general, we use one or more ADTs to specify the state information of an object. An object defined by a template can move from one state to another by LOTOS *events* defined in the body of the process definition which correspond to the *services* in the object model. Thus, a process definition together with its formal parameters specify the services (with their methods implemented as operations in the ADTs) and the attributes of the corresponding class template.

The Process Definition

The process defining a class template, called a *process template*, can use any combination of the LOTOS operators to specify the behaviour required (see Table A.2 in Appendix A). The state information is given by one or more ADTs which specify the operations required to support the services offered by the template.

An example of a process template using the choice operator, "`[]`", is as follows:

```
process Template[g](state: State_Template) : noexit :=
  ( g !selector_1 !Get_Id(state) ...;
    ...
    exit(state)
  []
    g !modifier_1 !Get_Id(state) ...;
```

```
    ...
    exit(F1(state))
  []
    ...
  ) >> accept update_state: State_Template in Template[g](update_state)
endproc
```

The process `Template` is defined recursively and uses gate `g` for synchronization with other processes. It communicates with other objects in the system through structured events. A structured event is composed of:

- a gate name for communication;
- a service name which plays the role of message name;
- the identifier of the object being called;
- a list of optional parameters.

The event:

```
g !selector_1 !Get_Id(state) ...;
```

is a structured event where `g` is a gate, `selector 1` is the message name which corresponds to an offered service in the object model, and `Get Id(state)` is an operation defined in the ADT `State_Template` and which gives the object identifier. This operation is required when we group the object identifier with other attributes in the same ADT. (We reserve the term *operation* to be used in the context of an ADT. In the context of a class template defined in an object model, we use the term *service*.)

The operator `[]` is the non-deterministic choice operator and `>>` is the enable operator. The behaviour expression `A>>B` means that on successful completion of process `A` we start execution of process `B`. The constructor `accept ... in` is used to pass values as we exit from one process and enable another.

The above template is offering a selector, given by the message name `selector 1`, and a modifier, given by the message name `modifier 1`. As a selector does not change the state of the object, **exit** returns the initial state given as a parameter of the process. On the other hand, as the modifier changes the state of the object, **exit** returns the value obtained by applying `F1` to the initial state. The operation `F1` is defined in the ADT `State_Template`.

Using the running example, let us consider the object model represented in Figure 3.2. Ignoring the services `create` and `remove`, for the time being, the template `Account` could be defined as:

```
process Account[g](this_account: State_Account) : noexit :=
 ( g !deposit !Get_Account_Number(this_account) ?m: Money;
   exit(Credit_Account(this_account, m))
 []
   g !get_balance !Get_Account_Number(this_account)
     !Get_Balance(this_account);
   exit(this_account)
 []
   g !withdraw !Get_Account_Number(this_account) ?m: Money;
```

```
    ( choice if_money: Bool []
      [if_money] -> g !rtn_withdraw !Get_Account_Number(this_account) !true;
                    exit(Debit_Account(this_account, m))
    []
      [not(if_money)] -> g !rtn_withdraw !Get_Account_Number(this_account)
                           !false;
                         exit(this_account)
    )
  ) >> accept update_account: State_Account in Account[g](update_account)
endproc
```

The structured event `g !deposit !Get_Account_Number(this_account) ?m: Money;` is the first action prefix expression in the behaviour expression

```
g !deposit !Get_Account_Number (this_Account) ?m: Money;
exit(Credit_Account(this_account, m))
```

and it denotes the offered service `deposit`. There are no required services from other objects in that behaviour expression, but if there were to be, they would appear after that action prefix.

The operations `Get Account Number`, `Credit Account` and `Get Balance` are defined in the ADT in which sort `State Account` is defined. The parameter `this account` represents the object's state information and is updated by the recursive call.

The generalized choice operator **choice**, used to specify the service `withdraw`, allows the introduction of non-determinism. Notice that by using this operator we can specify the two possible situations with the account, the account has sufficient funds and the account does not have sufficient funds, without querying the account's balance and without doing any calculations (as we will see when specifying ADTs).

Offering services as the alternative events of a choice expression is the most common LOTOS representation of a class template. It is not however necessary for all templates to look like `Account`. The structure and the operators depend on the behaviour we want to specify.

As a rule, we propose that the name given to a process template is the one used by the corresponding class template in the object model, with an initial upper case letter.

The ADT Definition

The arguments of a process are defined as ADTs. Given that LOTOS provides few built-in facilities, defining ADTs tends to be a time consuming task. Each operation is defined by one or more algebraic equations. Since we are in the analysis phase, and do not wish to become involved with design issues, it would be preferable to use ADTs where only a small number of simple equations are required. This can be achieved by defining ADTs which contain only the necessary information to allow the specification to be prototyped with state information and values to be passed during the communication. Thus, we are interested in the kind of information that is to be transferred between objects rather than the details of the algorithm by which the information is to be calculated within an object.

We build an ADT in the following way:

1. *Leave the modifiers without equations.* This treats them as constructors of the ADT and gives a record of the history of the events that have changed the object's state information.

2. *Define dummy equations for selectors when a particular result does not need to be returned.* More detail will be added in the design phase. A dummy equation does not query the state of the ADT and always returns the same constant value. It therefore adds no information that was not already in the signature. An equation must be given as otherwise a new constructor on the result sort would have been defined.

   The dummy equations are used in conjunction with non-determinism introduced in the process part, and it is there that the different possible situations are covered.

3. *Define equations for selectors that need to return a particular value.* The selector must be defined using an equation for each constructor.

The ADT that defines the sort `State Account` could be as follows:

```
type Account_Type is Account_Number_Set_Type, Money_Type, Balance_Type
 sorts State_Account
  opns Make_Account       : Account_Number, Balance -> State_Account
       Credit_Account     : State_Account, Money    -> State_Account
       Debit_Account      : State_Account, Money    -> State_Account
       Get_Balance        : State_Account           -> Balance
       Get_Account_Number : State_Account           -> Account_Number
       ...
  eqns forall a: State_Account, n: Account_Number, m: Money
    ofsort Balance
       Get_Balance(a)  = Some_Balance;
    ofsort Account_Number
       Get_Account_Number(Make_Account(n,m))   = n;
       Get_Account_Number(Credit_Account(a,m)) = Get_Account_Number(a);
       Get_Account_Number(Debit_Account(a,m))  = Get_Account_Number(a);
endtype
```

In `Account_Type` there is one constructor (`Make Account` which creates an account from its components), two modifiers (`Credit Account` which credits the account, and `Debit Account` which debits the account), and two selectors (`Get Balance` which returns an account balance and `Get_Account_Number` which returns an account number). For the constructors and the modifiers we give their signature and no equations. The selector `Get_Balance` does not need to return a particular value of balance (it is not important for us) and so it is defined with a dummy equation, always returning the value `Some Balance`. `Some_Balance` is a constant defined in the abstract data type `Balance Type`.

Since we use non-determinism in the process part, the use of dummy equations in the ADT does not exclude the study of the different possible situations. For example, we use the non-deterministic **choice** operator in process `Account` and, along with that, we explore the two possible situations which can happen: either there is enough money in an account or there is not enough money in an account.

`Get_Account_Number`, however, has to return a particular account number and so it is defined with an equation for each constructor.

The number of ADTs defined as parameters of the class template can vary. We would like to be able to incorporate all the attributes of an object in a single ADT. However, as we will see, inheritance of attributes and associations are better modelled as separate ADTs. We can also decide to model some attributes separately for reasons of reusability.

As a rule, we propose to name an ADT which defines an object state by using the template name followed by the suffix `_Type`, and let the sort name be the name of the template prefixed by `State _`. The operations of the ADT can be named differently from the services in the object model, or else with the same name followed by `_ADT`. The sort of auxiliary ADTs, such as `Balance_Type` and `Account_Number_Set_Type`, can be named without the term `State _`. The constructor is always named with the template name prefixed by `Make _`and the name of the operation that gives the object identifier always starts with `Get _`and follows with the identifier name.

3.3.3 Services

As noted previously, *service* is used in the context of an object model and a process, and *operation* is used in the context of an ADT. Also, we use the term *basic constructor* to denote a constructor in LOTOS that creates a value of an ADT from its components. `Make_Account` is an example of a basic constructor.

A service name in the object model appears as a message name in a LOTOS structured event in the process that defines the template. The method of a service is defined as a LOTOS behaviour expression and may include one or more operations in the ADTs given as arguments of the process template.

Whenever possible, the interaction between a server and a client is modelled by a single event as in the behaviour expression:

```
g !deposit !Get_Account_Number(this_account) ?m: Money;
exit(Credit_Account(this_account, m))
```

defined in template `Account`. The message name is `deposit` (it has the same name as the service in the object model). `Credit_Account` is the name of the corresponding operation on the sort `State_Account`. The whole behaviour expression is the *offered service* in LOTOS, and we can call it by its message name, in this case `deposit`.

The (first) action prefix `g !deposit !Get_Account_Number(this_account) ?m: Money;` denotes the signature of the offered service `deposit` [3]. (In Chapter 5 we give a formal definition of a service.)

When the specified behaviour is complex, a call/return pair of events must be used as in:

```
g !withdraw !Get_Account_Number(this_account) ?m: Money;
( choice if_money: Bool []
  [if_money] -> g !rtn_withdraw !Get_Account_Number(this_account) !true;
                exit(Debit_Account(this_account, m))
[]
  [not(if_money)] -> g !rtn_withdraw !Get_Account_Number(this_account) !false;
                     exit(this_account)
)
```

In this case, the signature of the offered service `withdraw` is the combination of the action prefix `g !withdraw !Get_Account_Number(this_account) ?m: Money;` and the return event `g !rtn_withdraw ....` (The return event always has the service name prefixed with `rtn` .)_

[3] Informally we can use the term *service* to denote *signature of the service*.

In order to perform a withdrawal, `Account` has to investigate whether or not the amount to be debited is less than (or equal to) the existing balance. The **choice** operator is non-deterministic and allows us to specify both the situation where there is enough money to debit the account (given by the guard `[if_money]`), and the situation where there is not enough money and therefore the debit is not allowed (given by the guard `[not(if_money)]`). Non-determinism is a useful mechanism in the analysis phase since it allows us not to compromise the decisions which may only be correctly made during the design phase. Following the guards, `Account` offers another event (`rtn withdraw`) for synchronization which returns information about whether or not the service was successful.

The message names are defined as constructor operations in a specific ADT called `Op_Names`, as follows:

```
type Op_Names is
  sorts Op_Name
  opns
    deposit, withdraw, get_balance, create, remove, ... : -> Op_Name
endtype
```

This discussion has referred only to the services an object offers to its environment or to external objects, but in order to accomplish them it may need to refer to other objects' services. The call to such a required service appears after the action prefix that defines the offered one. For example, `Counter Teller` offers in gate `t` the service `deposit cash`. In order to accomplish a deposit, `Counter Teller` must call the service `deposit` defined in `Account`, using gate `g` for synchronization. The appropriate part of the process definition of `Counter_Teller` is:

```
t !deposit_cash !id ?n: Account_Number ?m: Money;
  (* deposit_cash being on offer to the users *)
g !deposit !n !m;
  (* call deposit service defined in Account *)
```

where `id` is the identifier of the counter teller. This is needed to show the choice the client made in choosing this teller.

3.3.4 Attributes

As each attribute of a class template is modelled as a separate ADT, we could give a value of each one as a parameter of the template. However, we prefer to compose all the attributes in a single ADT and then use this one as the parameter of the process (just as we did for `Account`). This is not, however, always possible. If a subclass extends the attributes inherited from its superclass, the new attributes will be modelled as ADTs and given as extra parameters of the process template that defines the subclass. Similarly, an association between objects will be modelled as an attribute defined as an ADT and given as a parameter of the process defining the class template.

The basic constructor in an ADT defining an object state shows the components of a data value. The elements in its domain correspond to the attributes of the object. In the example given above, `Make_Account` is the basic constructor and `Account Number` and `Balance` are the two attributes of an account. These two attributes are defined in two separate ADTs. A basic constructor will not always have all the attributes in its domain as some may be implicit.

3.3.5 Classes

We propose to model a class in LOTOS by means of an *object generator*. An object generator is built from a template and it allows the creation of objects that share the same set of features.

Consider the following process definition of a simple object generator:

```
process Object_Generator[a] : noexit :=
  Template [a] ||| i; Object_Generator [a]
where
   process Template[a] : noexit :=
    (* some behaviour *)
   endproc
endproc
```

The interleaving operator `|||` indicates that the two processes `Template` and `Object Generator` are composed in parallel, but do not interact with one another. The internal event "`i`" is used to control recursive instantiation of `Object Generator` so that infinite recursion can be stopped.

In most real situations an object needs to be initialised when it is created and the initialisation operation should only be offered once for each object. We could do that inside this template as follows [Cla92b]:

```
process Object_Generator[a] : noexit :=
  Template[a] ||| i; Object_Generator[a]
where
  process Template[a] : noexit :=
    a !create; Template_1[a]
  where
    process Template_1[a] : noexit :=
     (* some behaviour *)
    endproc
  endproc
endproc
```

An alternative solution, which eliminates the need for the internal event and only uses two processes, is:

```
process Object_Generator[a] : noexit :=
  a !create;
  (Template[a] ||| Object_Generator[a])
where
  process Template[a] : noexit :=
    (* some behaviour *)
  endproc
endproc
```

Here, `Template` does not encapsulate the service `create`, letting it be offered by the generator. This is the way in which `create` should be regarded. Adopting this view, the definition of the class template is much simpler.

In the initialisation we can pass values that are then used to build the state information of an object. We adopt the rule that one of the values in the initialisation event is always

the object identifier. For each class of objects, we can define a set of identifiers given as a formal parameter of the corresponding object generator. The template's process definition includes a parameter giving the state of the object. Consider the following definition:

```
process Object_Generator[a](ids: Id_Set) : noexit :=
  a !create ?id_counter: Id ?init_val1: Value_Sort1 [id_counter notin ids];
  ( Template[a] (Make_Template(id_counter,init_val1))
  |||
    Object_Generator[a](Insert(id_counter, ids))
  )
where
  process Template[a](state: State_Template) : noexit :=
    (
      (* behaviour expressions with exit functionality *)
    ) >> accept state_modified: State_Template in Template[a](state_modified)
  endproc
endproc
```

The object generator holds the set of identifiers already allocated, `ids`. The selection predicate:

```
[(id_counter notin ids)]
```

guarantees that the new object identifier is different from all existing ones of the same sort. When `Object _Generator` is instantiated it offers synchronization with the event:

```
a !create ?id_counter: Id ?init_val1: Value_Sort1 [id_counter notin ids];
```

When an object is required, another object offers an event such as

```
a !create ?object_id: Id !val1;
```

Then, synchronization takes place and `Template` is instantiated causing an object to be created with some state information. Note that while the value `val1` is passed into `Object_Generator`, the object identifier is created using value generation. (In this example, the structured event that originates the creation of the object has only one optional parameter, `val1`. In other examples, several optional parameters may be required.)

In the case we are presenting, we are supposing that the set `ids` is initialised to be the empty set. For each new object created, its identifier is added to `ids`. In the banking system, a possible object generator for `Account` would be:

```
process Accounts[g](accs: Account_Number_Set) : noexit :=
  g !create ?acc_counter: Account_Number [(acc_counter notin accs)];
  ( Account[g](Make_Account(acc_counter,0))
  |||
    Accounts[g](Insert(acc_counter,accs))
  )
endproc
```

While the templates are named by using the name (in the singular) of the corresponding class template in the object model, object generators are named with the plural of the template's name.

By using object generators we are able to create, through the behaviour of the object generator, an infinite number of objects. There are however situations where we know the exact number of objects we need. If we only need one object of a given class during the

system's life an object generator is not required. Also, there are situations with composite objects where we may need to impose a fixed number of components. In this situation the instantiation of the template process would be done by explicitly calling that template's name. For example, if only one object in our application was required, the object would be created by:

```
Template[a](Make_Template(id1 of Template_Id, val1))
```

where `id1` is a specific identifier we assign to the object and `val1` is a value of sort `Value_Sort1`.

There is also the situation where we can impose an upper limit to the number of objects. In such a case, the object generator would hold the set (non-empty) of identifiers not yet allocated and the selection predicate would guarantee that the new object identifier is one of those.

3.3.6 Object Identity

In LOTOS, when we instantiate a process we must give a unique identity to each new object created. This is the purpose of an *object identifier*.

Object-oriented analysis methods propose that only attributes known from the real world should appear in the object model. In many situations object identifiers do not have a meaning in the real world and so, according to the above proposition, they should be added to the model in a later phase. This is a common procedure when dealing with informal and non-executable specifications, but it cannot be followed for formal executable specifications such as those in LOTOS, where objects are created dynamically during prototyping.

In the analysis phase, we are interested in defining a simple mechanism to be used to define distinguished object identifiers for each class template. This is achieved by defining the two special ADTs `Id_Type`, with the sort `Id`, and `Set_Id_Type`, which actualises the `Set` ADT already defined in the library. In order to have these two ADTs always available, we add them to the LOTOS library (see Appendix B).

As noted earlier, the object generator in its extended form requires a set of identifiers. Whenever object identifiers are needed for a given class, `Set_Id_Type` is instantiated. For example, for the class of objects `Account`, we instantiate it as follows:

```
type Account_Number_Set_Type is Set_Id_Type
   renamedby
   sortnames Account_Number     for Id
             Account_Number_Set for Set
endtype
```

`Account_Number` is the sort name of the object identifiers `id1, id2, ...` of an account. Recalling the template `Account`:

```
process Account[g](this_account: State_Account) : noexit :=
   g !deposit !Get_Account_Number(this_account) ...
 []
   ...
endproc
```

`Get_Account_Number`, defined in the ADT `Account_Type`, returns the appropriate account identifier.

Generating Identifiers

Whenever a new object is created we have to produce an identifier. The Open Distributed Processing model [ISO94] suggests four different ways to generate or allocate names (identifiers) for objects:

1. Allow the object to chose its own name, and ensure that it is suitably unambiguous;
2. Elect to use some information already known to identify the object unambiguously;
3. Allocate unique identifiers (e.g. numbers) to the objects, perhaps in the order in which they come into existence;
4. Some hybrid of the above.

For its simplicity, we choose the option 3 together with *value generation* which allows the introduction of uninstantiated variables.

For example, as presented in Section 3.3.5, an account number would be generated when the object generator `Accounts` offers for synchronization:

```
g !create ?acc_counter: Account_Number [(acc_counter notin accs)];
```

and some other object offers:

```
g !create ?acc_number: Account_Number;
```

When these two events synchronize, one value of the set of possible values of the sort `Account_Number` is ascribed to both `acc number` and `acc counter`. This value is constrained such that it is not already in the set `accs`. This technique is called *value generation* and it avoids defining an algorithm to generate the account number.

Value generation is supported by the SMILE simulator from LITE [4] [EW93] in the following way: it generates a symbol which may take any value from the set of possible values. This allows us to execute the specification symbolically rather than use particular values.

3.3.7 Objects

An object encapsulates its state and the algorithms which accomplish its behaviour. Any change in an object's state is only possible by means of interaction through a well defined interface with the environment of that object. An object is a member of a class and is created by instantiating a class template.

Creation (of an Object)

In cases where the operation that creates an object is offered to the environment, the operation `create` appears in the object model. This happens with `Account`, where a client can ask to open an account. In other cases, the creation operation does not appear in the object model. In our method, the operation `create` is not defined in the process template, but in the object generator. An object is created by instantiating the process which defines the class template. This can happen in two different ways:

1. If we want to create the objects dynamically, the instantiation occurs indirectly by sending a create message to an object generator;

[4] LITE is a set of tools for LOTOS produced as part of the LOTOSPHERE ESPRIT Project.

2. If we want to create the objects statically, the template is instantiated directly and an object generator is not required.

Consider that in the banking example, an unknown number of accounts are needed. Then, a new account is created by means of the object generator `Accounts` which offers:

```
g !create ?acc_counter: Account_Number [(acc_counter notin accs)];
```

and an instance of `Counter_Teller` would offer:

```
g !create ?acc_number: Account_Number;
```

in order to open an account. If only a single instance (or a small number) of account objects had been required, then we would not have defined the generator `Accounts` and each account object would be created by calling the template `Account` directly, for example:

```
Account[g](Make_Account(id1 of Account_Number, 0))
```

Deletion (of an Object)

A deletion operation may or may not appear in the object model. There are situations where we may want an object to "live forever", but there are others where we require explicitly that an object should be removed. For this, we define a `remove` service in the template that defines the object. In situations where inheritance is involved, this service must be defined in the process defining the subclass template.

The termination of an object is accomplished by terminating its LOTOS process. Termination of a LOTOS process is achieved by two LOTOS behaviour expressions: **exit**, representing the successful termination of the process; and **stop**, representing the abnormal termination of the process. Therefore we use **stop**.

To delete an object, the service `remove` in the object model is modelled as follows:

```
process Account[g](this_account: State_Account) : noexit :=
 ( ...
 []
   g !remove !Get_Account_Number(this_account);
   stop
 ) >> accept update_account: State_Account in Account[g](update_account)
endproc
```

Note that we are defining the `remove` operation in the template `Account` because we are ignoring the fact that an account is an abstract superclass, with subclasses. If we were taking this into consideration, this operation would be defined for each subclass.

State (of an Object)

The state of an object is given by the values of the parameters defined in the process template and by the services currently being offered by the object.

Consider that `Template A` offers the two services `service 1` and `service 3` and that after `service 1` has been called only `service 2` is on offer. Such a situation can be specified in LOTOS as follows [5]:

[5] To simplify the problem, we are considering that the services do not change attribute '`s`'.

```
process Template_A [g] (s: State_Sort) : noexit :=
  ( g !service_1;
    g !service_2;
    exit(s)
  []
    g !service_3;
    exit(s)
  ) >> accept update_state: State_Sort in Template_A[g](update_state)
```

Here, the state of an object of the class template `Template A` is given by the value of `s` together with the services on offer at each moment.

For simplification, we could define the state of an object as being determined by the values of its attributes. In such a situation, the state would always be given by the parameters of the process, but the definition of `Template A` has to be changed to:

```
process Template_A [g] (s: State_Sort, b: Bool) : noexit :=
  ( [not(b)] -> g !service_1;
                exit(s, true)
  []
    [b] -> g !service_2;
           exit(s, false)
  []
    [not(b)] -> g !service_3;
                exit(s, false)
  ) >> accept update_state: State_Sort, update_bool: Bool
   in Template_A[g](update_state, update_bool)
```

`b` would normally be defined within `State Sort`. Here we are giving it explicitly as a separate argument of the process only to simplify the explanation.

In LOTOS, these two definitions of `Template A` are similar in practice and it is trivial to transform one into the other. However, it is easier to explain 'state of an object' in mathematical terms if we take the second definition. This is the reason why we use this definition in Chapter 5. In ROOA we prefer to adopt the first definition, because it produces a more compact specification.

Behaviour (of an Object)

A *behaviour* describes the order in which the (LOTOS) offered services of that object can occur and the changes in the object's state. Behaviours are required to define *sequencing rules*, the possible choice of services at any given time, and, for more complex behaviours, *concurrency rules* [ISO94]. The state of an object can restrict the services that can be offered by the object at a given time as LOTOS (structured) events can have guards. In the ROOA context, a structured event (or just event for short) corresponds to the signature of a service.

The behaviour of an object is given in LOTOS by the externally visible behaviour and by the internally invisible behaviour. The externally visible behaviour is specified by structured events which occur when the object synchronises with other objects in the system. In terms of ROOA, this means that one object is calling a service of another object. The structure of the event which corresponds to calling a service (in the calling object) is equivalent to the structure of the event that defines the signature of the called

service (in the called object). The internal, invisible behaviour is specified by structured events only defined internally to the called object and so are invisible to the object's environment.

In summary, the behaviour of an object is given by a collection of offered services (with signature and methods, as we will see in Chapter 5) with a set of constraints on the order in which they may occur. In Chapter 5 we discuss the behaviour of class templates and objects in more detail.

3.3.8 Message Connections

In LOTOS two or more processes communicate via event synchronisation by using gates. However, as message passing in the object-oriented paradigm involves two objects, we restrict communication in LOTOS to be defined between two process instances which synchronise at a common gate on an externally visible event.

An object can behave as a *client*, as a *server*, or both. Client objects send messages to server objects which may or may not return an answer. A server normally offers all its services at a single gate. If it also acts as a client, then it uses separate gates to communicate with its servers.

Communication is achieved by synchronizing on a structured event of the form:

⟨*gate name*⟩ ⟨*message name*⟩ ⟨*object identifier*⟩ ⟨*optional parameters*⟩

For example, a `Counter_Teller` can send a message to `Account` asking for a deposit:

```
g !deposit !acc_number !amount;
```

and an instance of `Account` synchronizes with this event by offering:

```
g !deposit !Get_Account_Number(this_Account) ?m: Money;
```

Value matching of `acc_number` and `Get_Account_Number(this_account)` is used to ensure correct synchronization. Value passing is used to pass the value `amount` to the variable `m`. Although a client must know the identity of the server, a server can service many clients without knowing their identity.

The client gives the service (message) name, the server object identifier and, optionally, some parameters. In order to give an answer, the server can either accomplish one of its methods, send a message to another object, or both. Sometimes the request and the answer can be specified in LOTOS as a single behaviour expression in each object (as happened above). In this case, the entire communication is an atomic event. Another example is when a `Counter_Teller` sends a message to `Account` asking for an account balance:

```
g !get_balance !acc_number ?balance: Money;
```

and an instance of `Account` synchronizes with this event by offering:

```
g !get_balance !Get_Account_Number(this_account) !Get_Balance(this_account);
```

In general, the server may not be able to give the result immediately in which case the client must offer a second synchronization event to receive the server's result. These events form a non-atomic action which can be interpreted as a form of remote procedure call. In this situation, the second synchronization should not be understood as if the server object was now behaving as a client of the initial client object. Neither the "call event" nor the "return event" includes the identifier of the object which initiated the communication.

An example is when `Counter_Teller` sends a message to withdraw money. This requires a call (from `Counter_Teller` to `Account`) and a return event (from `Account` to `Counter_Teller`).

```
g !withdraw !acc_number !amount;
g !rtn_withdraw !acc_number ?ok: Bool;
```

Both events use the value of the account number (i.e. the identity of the server) to ensure correct synchronization. `Account` "accepts" the call and receives the parameter `amount`. Then it carries out the method `withdraw` and then returns the result to the client `Counter_Teller` in one of two alternative events:

```
g !withdraw !Get_Account_Number(this_account) ?m: Money;
( choice if_money: Bool []
  [if_money] -> g !rtn_withdraw !Get_Account_Number(this_account) !true;
                exit(Debit_Account(this_account, m))
[]
  [not(if_money)] -> g !rtn_withdraw !Get_Account_Number(this_account) !false;
                     exit(this_account)
```

The discussion presented above shows how two objects which belong to different classes communicate. These two objects are connected by a message connection in the object model. Usually, as we have seen, if an object wants to initiate a communication then it has to receive, from its environment or the external world, the object identifier of the object with which it wants to communicate.

It is possible that two objects belonging to the same class template need to communicate. We discussed in Section 3.2.16 how unary associations give the capability of communication between two objects of the same class. In LOTOS, the only way to specify such a communication is by creating a *channel* of communication. This channel is an object, which serves as an intermediary between the two original objects (see Figure 3.3).

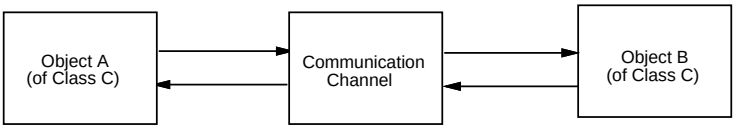

Figure 3.3: Two objects of the same class communicate via a channel

Having created this object, we converted the 'unary' communication into a 'binary' communication, and so we can follow the general procedure. Therefore, a channel is defined as a process, which synchronizes with the process which defines the original class template.

3.3.9 Specifying Inheritance with LOTOS

As LOTOS was developed before object-oriented techniques became widely accepted, inheritance is not directly supported. However, by using the standard LOTOS constructs, incremental inheritance can be represented in a straightforward way. In this section we only deal with incremental inheritance. Pure extension, where there is no redefinition or deletion of services, does provide behavioural inheritance. For further discussion on

behavioural inheritance and on how the conditions given in Section 3.2 must be verified in a LOTOS specification, see [Cla94c, CM94, MC93a].

To be able to specify inheritance in LOTOS (extension and redefinition of services and extension of attributes), the superclass has to be defined with **exit** functionality [Rud92]. The reason is that, within the specification of a subclass, many of the offered services are provided by invoking the superclass. After a service defined in the superclass has been handled, all the services offered by the subclass must again be on offer. We must therefore exit from the superclass so that the subclass, and not the superclass, is invoked recursively. This means that all superclasses in LOTOS are abstract class templates. Although this is required by the syntax and semantics of LOTOS, Hürsch has discussed the advantages of superclasses always being abstract [Hür94]. (Later in this section we show how a concrete class template can be created from an abstract superclass.)

Considering *Fn(x)* to be any function involving x and defined as an operation in the ADT which defines the sort of x, the superclass would take the form:

```
process Superclass[g](state: State_Sort) : exit(State_Sort) :=
  g !selector_1 !Get_Id(state) ... ;
  ...
  exit(state)
[]
  g !modifier_1 !Get_Id(state) ... ;
  ...
  exit(F1(state))
[]
  g !modifier_2 !Get_Id(state) ... ;
  ...
  exit(F2(state))
endproc
```

We can create the subclass `Extended Class` based on that `Superclass` which is extended to offer more services:

```
process Extended_Class[a](state: State_Sort) : noexit :=
  ((Superclass[a](state)
      >> accept update_state: State_Sort in exit(update_state)
   )
  []
    a !modifier_3 !Get_Id(state) ... ;
    ...
    exit(F3(state))
  ) >> accept update_state: State_Sort in Extended_Class[a](update_state)
endproc
```

The first occurrence of the operator **accept ... in** is not needed, as the result of `Superclass` is of the same sort of the result given by `Extended Class` and the name of the parameters of both templates are the same. However, we can leave it there so that the skeleton of the subclass template takes a more general form.

Note that the subclass `Extended Class` could also be extended in the number of gates, if this was necessary to define the new services.

Rudkin presents a rigorous approach to how inheritance can be introduced in LOTOS, and describes the problems with self referencing (when the superclass has **noexit** functionality) [Rud92].

If a redefinition of one or more services is required, the idea is to "eliminate" them first and then create them with the necessary differences. To accomplish this it is necessary to have an auxiliary superclass where the services that are going to be directly inherited are defined. Supposing that we wanted to redefine the service `modifier 2`, we specify auxiliary class `Modifier Class` with the services we want to keep and use it as follows in the definition of the new class `Redefined Class`:

```
process Modifier_Class[a](state: State_Sort) : exit(State_Sort) :=
  a !selector_1 !Get_Id(state) ... ;
  ...
  exit(any State_Sort)
[]
  a !modifier_1 !Get_Id(state) ... ;
  ...
  exit(any State_Sort)
endproc
```

where `any` is a LOTOS keyword and can be used with any type, predefined or not.

```
process Redefined_Class[a](state: State_Sort) : noexit :=
  ( (Superclass[a](state) |[a]| Modifier_Class[a](state))
   []
     a !new_modifier_2 !Get_Id(state) ... ;
     ...
     exit(F2a(state))
   []
     a !modifier_3 !Get_Id(state) ... ;
     ...
     exit(F3(state))
  ) >> accept update_state: State_Sort in Redefined_Class[a](update_state)
endproc
```

As before, `modifier 3` is an added service.

So far, we have been discussing extension and redefinition of services. But, how can we create a subclass which extends the state information of its superclass? As the attributes are defined in the abstract data type part, it seems that incremental modifications in the attributes of the object should be done there. There are however some complications. We can use the ACT-ONE language to extend (and combine, and rename) abstract data types, but only in what concerns the operations defined in the ADT. If we want to extend the number of components, then the functions defined in the initial abstract data type cannot be inherited, since the constructor operations need to "know" about all the data components of the structure. The solution is to add more ADTs as parameters of the class template that defines the subclass, although this gives us a broken structure for the state information of the object.

Taking `Superclass` and `Modifier Class` defined above, and supposing we want to define a subclass which extends the superclass state and redefines the service `modifier 2`, the process template `Redefined Class` would take the form:

```
process Redefined_Class[a, b]
     (state: State_Sort, ext_state: Ext_State_sort) : noexit :=
  ( (( Superclass[a](state)
     |[a]|
       Modifier_Class[a](state)
     )
       >> accept update_state: State_Sort in exit(update_state, ext_state)
    )
   []
     a !new_modifier_2 !Get_Id(state) ... ;
     ...
     exit(F2a(state), F2b(ext_state))
   []
     b !modifier_3 !Get_Id(state) ... ;
     ...
     exit(F3a(state), F3b(ext_state))
  ) >> accept update_state: State_Sort, ext_update_state: Ext_State_Sort
       in Redefined_Class[a, b](update_state, ext_update_state)
endproc
```

Now, the state information of objects of the class template `Redefined Class` is the pair of ADTs `(state, ext_state)`.

In the object model represented in Figure 3.2, `Cheque Account` and `Savings Account` are identified as subclasses of the superclass `Account`. In Section 3.3.2, `Account` was defined with **noexit** functionality, but if it is to be used as a superclass, its functionality has to be changed to **exit**:

```
process Account[g](this_account: State_Account) : exit(State_Account) :=
   g !deposit !Get_Account_Number(this_account) ?m: Money;
   exit(Credit_Account(this_account, m))
 []
   g !get_balance !Get_Account_Number(this_account)
     !Get_Balance(this_account);
   exit(this_account)
 []
   ...
endproc
```

The superclass can then be extended to create a `Cheque Account` subclass. The new class template inherits the properties of the superclass and defines the new operation `print_mini_statement`.

```
process Cheque_Account[g](this_account: State_Account) : noexit :=
  ( Account[g](this_account)
  []
    g !print_mini_statement !Get_Account_Number(this_account) !this_account;
    exit(this_account)
  ) >> accept update_account: State_Account
    in Cheque_Account[g](update_account)
endproc
```

Both kinds of account (savings account and cheque account) have identifiers of sort `Account_Number` in order to inherit from the same superclass. This is why we divided

the identifiers defined in the ADT `Id Type` into several groups (see Appendix B). Therefore, the ADT `Account_Number_Set_Type` needs to be changed in order to include this information:

```
type Account_Number_Set_Type is Set_Id_Type
   renamedby
   sortnames Account_Number     for Id
             Account_Number_Set for Set
   opnnames  Is_Cheque_Acc  for First_Set
             Is_Savings_Acc for Second_Set
endtype
```

`Is_Cheque_Acc` and `Is_Savings_Acc` establish the set of identifiers which can be used to create cheque accounts and savings accounts, respectively.

An object generator for `Cheque_Account`, for example, would be:

```
process Cheque_Accounts[g](accs: Account_Number_Set) : noexit :=
  g !create ?acc_counter: Account_Number !cheque
     [(acc_counter notin accs) and Is_Cheque_Acc(acc_counter)];
  ( Cheque_Account[g](Make_Account(acc_counter, 0))
  |||
    Cheque_Accounts[g](Insert(acc_counter,accs))
  )
endproc
```

`!cheque` is required to give the type of account we want to create. The object generator holds the set of identifiers already allocated and the selection predicate:

```
[(acc_counter notin accs) and Is_Cheque_Acc(acc_counter)];
```

imposes the condition that the new object identifier is different from all existing ones and `Is_Cheque_Acc(acc_counter)` guarantees that the new object identifier belongs to the correct subrange of `Account_Number`.

We defined the superclass `Account` in the object model not to have any specific instances, and so it does not need an object generator. However, if we change the requirements in order to allow the creation of objects of the superclass, a new process would have to be created. Let us call this process `Special_Account`:

```
process Special_Account[g](this_account: State_Account) : noexit :=
  Account[g](this_account)
    >> accept update_account: State_Account
    in Special_Account[g](update_account)
endproc
```

The inheritance hierarchy would now be as depicted in Figure 3.4.

If several instances of `Special Account` were required, we would define an object generator.

3.3.10 Abstract Class Templates

A process defining an abstract class template does not have any instances and is only used in the definition of processes which define subclass templates. Process `Account` defined in Section 3.2.15 with **exit** functionality is an example of an abstract class template in LOTOS.

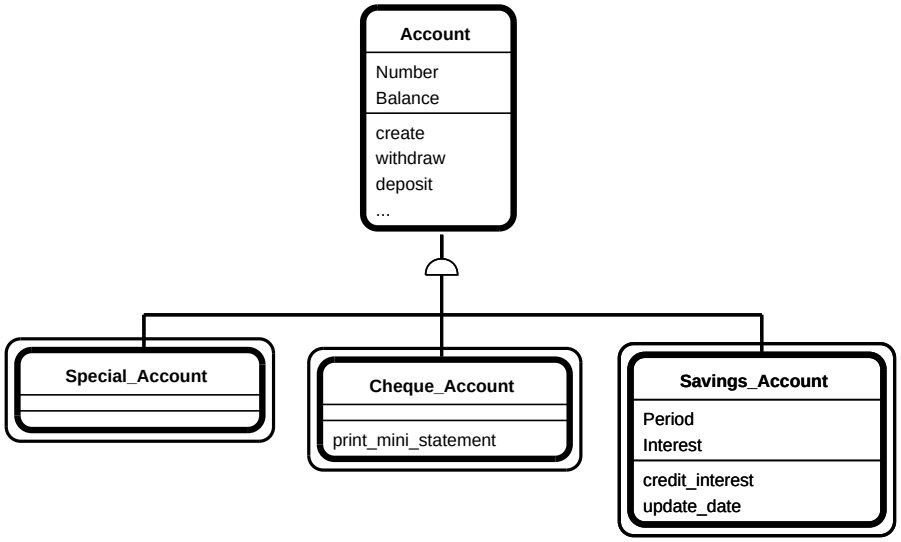

Figure 3.4: `Special_Account` as a concrete class template of the abstract superclass `Account`

3.3.11 Conceptual Relationships

We represent conceptual relationships (associations) in LOTOS as arguments in the process defining the class template. These arguments are ADTs which represent either the identifier of an object or a set of identifiers, depending on the cardinality of the association. Since we are in the analysis phase, we do not want to decide how certain properties of the system should be designed and then implemented. However, if we are creating executable specifications, we have to model associations in order to be able to simulate the results, but it does not mean that we have to make final decisions at this stage. Later, in the design, we will decide the best way to implement an association. It may well be that we may represent an association as a new object.

We propose that any relationship involving a superclass will be inherited by the objects of its subclasses. Therefore such relationships are modelled in the template that defines the superclass.

Binary Relationships

One-To-One

A *one-to-one* relationship is modelled in each object as an attribute which is the required object identifier. If the minimum of the cardinality is *zero* we can use a set of identifiers, instead of the identifier itself. The empty set gives us a simple way of dealing with optional relationships.

In the following examples we only show the relationship being modelled in one of the objects. To model it in both objects a similar technique should be applied to the second object.

Suppose there is an optional 1:1 relationship between a cheque account and a card. The template `Cheque_Account` would be:

```
process Cheque_Account[g](this_account: State_Account,
                          cards: Card_Number_Set) : noexit :=
  ((Account[g](this_account)
     >> accept update_account: State_Account in exit(update_account, cards)
   )
  []
    g !print_mini_statement !Get_Account_Number(this_account) !this_account;
    exit(this_account, cards)
  []
    ...
  ) >> accept new_account: State_Account, cards: Card_Number_Set
    in Cheque_Account[g](new_account, cards)
endproc
```

and the object generator must be changed to initialise the parameter `cards` of the template `Cheque_Account`:

```
process Cheque_Accounts[g](accs: Account_Number_Set): noexit :=
  g !create ?acc_counter: Account_Number !cheque
     [(acc_counter notin accs) and Is_Cheque_Acc(acc_counter)];
  ( Cheque_Account[g](Make_Account(acc_counter, 0), {} of Card_Number_Set)
  |||
    Cheque_Accounts[g](Insert(acc_counter,accs))
  )
endproc
```

`{} of Card_Number_Set` represents the empty set.

If the relationship was mandatory, rather than optional, it would mean that for each account there must be one card. Therefore, instead of the empty set `{} of Card_Number_Set`, one card identifier of the sort `Card_Number` is needed. In the above example, by initialising the object with an empty set, we can create an account and later on create a card, if necessary. However, if the association is mandatory, then at the time we create an account, we must create the corresponding card. Thus:

```
process Cheque_Accounts[g, cd](accs: Account_Number_Set) : noexit :=
  g  !create ?acc_counter: Account_Number !cheque
     [(acc_counter notin accs) and Is_Cheque_Acc(acc_counter)];
  cd !create ?card_nr: Card_Number !acc_counter;
  ( Cheque_Account[g](Make_Account(acc_counter, 0), card_nr)
  |||
    Cheque_Accounts[g](Insert(acc_counter,accs))
  )
endproc
```

The gate `cd` is used to communicate with the `Card` object generator (which we are not showing here). The value `acc _counter` would only be passed if the association was bidirectional.

One-To-Many

A *one-to-many* association is modelled as an attribute that has the value of the object identifier in the *many* side (contained object) and as an attribute that is a set of object

identifiers in the other side (container object). Again, optional relationships are modelled by using a set of identifiers instead of a single identifier.

In the object model depicted in Figure 3.2, `Cheque Account` has a one-to-many association with `Card`. This case would be dealt with using a set of cards in the same way as the optional *zero-to-one* association we studied above.

Many-To-Many

A *many-to-many* association can be transformed into two one-to-many associations by creating a third object and using it to relate the other two objects. This is the way we handle this when dealing with relational databases. Here, for simplicity, we do not use that third object, modelling a many-to-many relationship as two one-to-many relationships, but in a way that each one appears in each object as an attribute that is a set of object identifiers.

In the banking system example, `Cheque Account` has a many-to-many relationship with `Standing Order`. We must add to `Cheque Account` the parameter `sos` of sort `SO Number Set`. The value of `sos` is the set of standing order numbers associated with that account. Any time a standing order is created, its identifier should be given to the corresponding account. (This is supposing that accounts know about standing orders. It could be that only standing orders had to know about accounts.)

The template `Cheque Account` with the extra argument `sos` is given below:

```
process Cheque_Account[g](this_account: State_Account,
                      cards: Card_Number_Set, sos: SO_Number_Set) : noexit :=
  ((Account[g](this_account)
     >> accept update_account: State_Account
        in exit(update_account, cards, sos)
   )
  []
    ...
  ) >> accept update_account: State_Account, cards: Card_Number_Set,
              sos: SO_Number_Set
    in Cheque_Account[g](update_account, cards, sos)
endproc
```

The object generator now has to instantiate the process template with one more parameter:

```
process Cheque_Accounts[g](accs: Account_Number_Set) : noexit :=
   ...
  ( Cheque_Account[g](Make_Account(acc_counter, 0),
                      {} of Card_Number_Set, {} of SO_Number_Set)
  |||
    Cheque_Accounts[g] ...
  )
endproc
```

In this case, a standing order knows about two accounts (the one which is going to be credited and the one which is going to be debited). Because we know the cardinality of the association in standing order/cheque account direction and also the accounts involved

in the association when a standing order is created, we can use the two account identifiers separately, instead of giving a set with the two identifiers as elements of that set:

```
process Standing_Orders[a, b, c](sos: SO_Number_Set) : noexit :=
  a !so_create ?n1: Account_Number ?n2: Account_Number ?bk: Bank_Name
    ?m: Money ?so_counter: SO_Number [so_counter notin sos];
  ( Standing_Order[a, b, c](Make_SO(so_counter, n1, n2, bk, m))
  |||
    Standing_Orders[a, b, c](Insert(so_counter, sos))
  )
endproc
```

Unary Relationships

Unary relationships are modelled as an identifier (or set of identifiers) in the process template. There are, however, situations where they can be seen as *is-a* (generalization/specialization) relationships. In object-oriented development, inheritance is an important concept which usually comes to light early in the development process. However, there are situations where this concept does not show up clearly. Consider the example of a company with its employees. It could be useful to define a relationship `manager` in the `Employee` object that relates an employee with his or her manager. If there is a significant difference in terms of behaviour or in terms of structure between the concept "employee" and the concept "manager", we should create a superclass `Person` and the two subclasses `Employee` and `Manager`. Otherwise, we add to `Employee` an attribute that gives the identifier of the manager who is also an employee. This would be done by adding to the template another argument which gives the manager object identifier.

Relationships with Values

Relationships can hold values. We could define a pair (or set of pairs) where the first component is the identifier of one of the objects and the second is the value and then give this information to one (or both) of the objects. However, a simple solution is to create a new object which holds the value and also the identifiers of the objects involved in the association.

For example, suppose an application deals with a stock of products and that we have to keep information about the suppliers of the products. Now, let us define the relationship `is_supplied` between `Product` and `Supplier` in which a supplier supplies many products and a product is supplied by a single supplier. Moreover, a client needs to know the quantity of a given product that a supplier supplied. The `quantity` attribute does not belong either to `Product` or `Supplier` individually, but to both, i.e. to the relationship. We then create a new object, called `Supply` which would be defined as follows:

```
process Supply[g](this_quantity: Quantity_Sort,
                  obj1_id: Object1_Id, obj2_id: Object2_Id) : noexit :=
  ...
endproc
```

Alternatively the parameters `this quantity`, `obj1 id` and `obj2 id` could have been defined as part of a single ADT.

Notice that we are supposing that the identifier of an object of this template is the pair `obj_id1`, `obj_id2`.

3.3.12 Composition and Decomposition

Objects are combined to form composite objects by using the LOTOS enabling, interleaving and parallel operators.

The characteristics of a composite object are determined by (a) the objects that are combined; and (b) the way they are combined.

3.3.13 Subsystems

A subsystem is only created to structure the system, helping us manage complexity. It does not add any extra functionality to the system. We model subsystems by using the LOTOS interleaving and parallel operators.

In our banking system, we create the subsystem `Financial Instruments`, built from the class templates `Cheque` and `Standing Order`:

```
process Financial_Instruments[ob, cs, ba] : noexit :=
  ( Cheques[ob, cs, ba]
  |||
    Standing_Orders[ob, cs, ba]({} of SO_Number_Set)
  )
endproc
```

where `Cheques` and `Standing Orders` are object generators.

A subsystem has a name (e.g. `Financial Instruments`), but it has no state or identifier (the process `Financial Instruments` has no arguments). The name is only needed when joining all the pieces of the system. It will never be used by any object (an account, for example) to communicate with a component (a cheque or a standing order, for example). This communication is direct, without the subsystem's knowledge.

During our experiments with ROOA, we discovered rules by which class templates can be grouped to form subsystems. Some of these rules were imposed by LOTOS, which is not the right reason for grouping, but they make sense as a rationale. We present them in Chapter 6.

3.4 Conclusions

This chapter described the representation of object-oriented analysis concepts in LOTOS. This includes the main concepts of: class template, class, object, communication between objects and inheritance. A class template defines the common characteristics of objects of the same kind and is modelled with a LOTOS process definition and one or more ADTs which represent the attributes. A class is the set of objects instantiated from a given class template. We extend this idea and propose object generators to create objects from a given class template. An object is a member of a class and is created by instantiating a class template.

As specifying ADTs is usually a lengthy and tedious task, we propose a simpler way to accomplish this job. The basic idea is that modifiers are left without equations, and selectors are defined with dummy equations, if they do not need to return a particular value, and with proper equations, if they need to return a particular value.

Communication between objects (message connections) is modelled by two LOTOS processes synchronizing on a structured event. During synchronization the two processes

may exchange data. Complex object interactions may be built out of simpler interactions, by using the LOTOS parallel operators.

Another main concept discussed in this chapter is that of inheritance. We model inheritance by using technical features of LOTOS, namely superclasses with exit functionality.

Chapter 4

Further Concepts: Complex Objects

4.1 Introduction

In entity-relationship models, the term aggregation is used to describe the relationship between an entity and its attributes. Each attribute is a component of an entity, and the entity is an aggregate of its attributes. We are interested in a broader definition of aggregation which describes the relationship between objects and allows a more complex object to be formed from the combination of simpler objects. The complex object is called the *aggregate* object and the simpler objects are called *object components*, or just *components*. Aggregates are described by *part of*, *whole part*, *component of*, or *consists of* relationships.

We have a broader view of aggregation than some other authors. An important characteristic of our aggregates is that they may add behaviour to the behaviour defined in their components. Moreover, components within the aggregate may communicate between each other without the aggregate's 'knowledge'. The services defined in a component may or may not be on offer by the aggregate. We classify aggregation according to whether or not the components are visible to the other objects in the system and also whether or not the number of components can vary in time.

We make a clear distinction between aggregates and *subsystems*. While an aggregate is an object, a subsystem is only a group of objects, without individual identity.

Although, in the past few years, researchers have paid special attention to aggregation, there is no standard definition of what an aggregate is. This chapter reviews the more common views of aggregation and presents our view, proposing a set of properties that aggregates should satisfy. Finally, it shows how ROOA uses LOTOS to model aggregates.

4.2 The Role of Aggregation

Aggregation is a relationship between several objects which allows a more complex object, the *aggregate*, to be built from a combination of simpler objects, the *components*. Both aggregates and components are objects; they have an identity, they offer services according to a certain behaviour and they have state information which records the results of their

services. The behaviour and state information of the aggregate is given by a combination of the behaviour and state information of its components and by extra data and functionality.

Aggregation with hidden components is a kind of abstraction. There are other kinds, generalization being an example. Abstraction is the suppression of detail about an object, except for that relevant to the immediate purpose. Other kinds of abstraction can be applied during implementation where a concept is described by parts, but none of these parts is a true object. Implementation abstraction hides implementation detail from the user. "Real-world" abstraction, such as aggregation and generalization, is useful when thinking about the real world. It defines a more abstract (higher level) object which is useful for our understanding of the problem. This sort of abstraction helps in controlling the size and complexity of large systems during development and, if a system is developed using levels of abstraction, the resulting product is less difficult to understand.

It is important to distinguish between the role of an aggregate and that of a grouping of objects, such as *subsystems*. We believe that an aggregate must be a representation of an entity from the real world and it may have information of its own, for example the number of components. In the more interesting case, the aggregate manages the interaction between its components and the rest of the objects in the model. Whereas subsystems may be used during analysis, and may have no implementation consequences, aggregates are objects which have first class status in the system and which usually appear in the final implementation.

A subsystem is not an object, it is a grouping of logically related objects. The behaviour of a subsystem is given by the behaviour of its components. Each component communicates directly with objects outside the scope of the subsystem, as if the subsystem did not exist. Wirfs-Brock *et al.*, for example, describe a subsystem as an analysis construct which does not survive in the implementation [WBWW90, p30]:

> A *subsystem* is a set of classes (and possibly other subsystems) collaborating to fulfil a set of responsibilities. Although subsystems do not exist as the software executes, they are useful conceptual entities.

These subsystems are groupings of objects useful for understanding a problem, but they may or may not describe a concept from the real world. Aggregates, on the other hand, describe complex concepts from the real world and so they are named objects.

The role played by aggregation varies from author to author. Rumbaugh *et al.* define it as a stronger form of association (conceptual relationship) [RBP $^{+}$91]; Smith and Smith stress its importance as a mechanism to introduce abstraction [SS77]; and Hartmann *et al.* [HJS92] use aggregation as a structuring mechanism and define its formal semantics, showing how it is supported by the specification language TROLL [JSHS91].

We see aggregation as a mechanism for structuring a large system into different levels of abstraction, in which each higher level object is described in terms of simpler objects. As we will see, it uses abstraction, information hiding and encapsulation as basic techniques.

4.3 Combination of Objects: Another View

Although aggregation seems a simple concept, a standard definition does not exist (see e.g. [Bla93, Civ93, HJS92, Ode94]).

As we are interested in modelling aggregation formally, this section discusses the view supported by Hartmann *et al.* who define a formal semantics for composite objects [HJS92]. According to them, the combination of objects to form a more complex object (such as an aggregate) is modelled by *structure-preserving* mappings between objects, also called *object morphisms*. A special case of object morphism is the *object embedding morphism* which describes a complex object ob_1 encapsulating an object ob_2. The set of life cycles of the encapsulated object ob_2 must be preserved and events of ob_1 must use events of ob_2 to modify ob_2's state. This latter requirement is captured by the concept of *calling*. If an event e_1, in ob_1, calls an event e_2, in ob_2, then whenever e_1 occurs, e_2 also occurs.

Hartmann *et al.* state that an object embedding morphism between objects ob_1 and ob_2, assuming that ob_2 is embedded in ob_1, has to satisfy the two following conditions:

1. The events of ob_2 are included in the set of events of ob_1. For the life cycle of the (composite) object ob_1, if we constrain a life cycle to the events of the embedded (or part) object, we have to obtain a valid life cycle of the embedded object.

2. The attributes of ob_2 are included in the attributes of ob_1. For observations of ob_1 projected to the attributes of the part object, we must obtain the same observation as for applying the observation mapping of part object ob_2 to a life cycle of ob_1 restricted to the events of ob_2.

An aggregate object that satisfies these two rules is the coproduct of its components. Such an object does not add any behaviour to the behaviour already defined in its components. Most object-oriented analysis methods argue that the interesting situation appears when the aggregate has properties of its own together with the properties of its components [CY91a, dCLF93, RBP+91].

Imposing the condition that every event defined in an object component has also to be defined in the aggregate forbids a component to have services hidden from the aggregate. Such hidden services would allow, for example, communication between components without the aggregate's intervention. Hence, the two rules forbid the communication between object components defined at the same level of the hierarchy and in the scope of the aggregate.

We want our aggregates to have extra functionality which is unknown to their components. Moreover, we want to allow communication between object components without the aggregate's knowledge. Therefore we permit services defined in a component, and which are only needed by another component, not to be on offer by the aggregate. For these reasons our aggregates are different from those of Hartmann *et al.*

4.4 Transitivity

As a component is an object, it may happen that this object is itself a complex object, composed of other objects. The object at the top level of the hierarchy need not know that one or more of its components are aggregates. This leads to the issue of transitivity. While some authors [dCLF93] explicitly say that aggregation is not transitive, other authors [RBP+91] define transitivity as one of the most important properties of aggregates.

Transitivity can be discussed according to the semantics of the aggregation relationship. Let us suppose that aggregate ob_1 has the component ob_2 which has a component ob_3. It

is the case that if ob_2 is part of the internal structure of ob_1 and ob_3 is part of the internal structure of ob_2, then ob_3 is part of the internal structure of ob_1. However, transitivity is lost when aggregation relationships with different semantics are involved. For example, I am part of a research group and my arm is part of me, but my arm is not part of my research group.

Message connections, for example, are not transitive. If ob_1 has a message connection with ob_2 and ob_2 has a message connection with ob_3, this does not mean that ob_1 has a message connection with ob_3. If this was not so, information hiding and encapsulation would not be available. Nevertheless, indirect communication is possible. When the top level object, object ob_1, requests a service from one of its components, object ob_2, it may well be that ob_2 delegates part of the service to one of its components, object ob_3. However, ob_1 does not need to know the ob_2-services which correspond to calls of ob_3-services. We can say that, in general, ob_1-services call ob_2-services which call ob_3-services. This is discussed by Rumbaugh *et al.* [RBP+91] as *propagation of operations* and is what Hartmann *et al.* [HJS92] call *event-calling.*

4.5 Classifying Aggregation

We describe an aggregate in terms of its components. We allow a component to be hidden (non-shared) from other objects or shared by other objects. We use the term *aggregation* to refer to the relationship between the aggregate and its components. As a conceptual relationship, aggregation has cardinality. We show cardinality at the end points of the relationship with an amount (k) or a range (n, m) [1].

A hidden component is not visible to other objects in the system and so it can only communicate with the rest of the system through the aggregate, although it can interact with other components in the same aggregate. An aggregate defines the scope of its hidden components and encapsulates each of them.

A shared component can be accessed by other objects in the system. For each shared component, an aggregate has an attribute holding the object identifier of that component.

A component encapsulates its own state and behaviour, in the sense that its state may only be changed using the services defined in its interface. The interaction between an aggregate and its components and among components is via communication [MC93c].

So far, we have been talking about aggregates and components as being objects. We will from here on talk about aggregate classes, component classes, aggregate objects and component objects. However, in situations where the meaning is understood by the context, we may use only the terms "aggregate" and "component".

In the object model of OOA methods the aggregate is represented by the class template at the top level of the hierarchy and the components are the class templates at the bottom (see Figure 4.1).

For our work, we allow aggregates with either a static or dynamic number of components, but we restrict the number of component classes to be constant. The structure of an aggregate with a constant number of components is defined at requirements-specification time and its composition never changes, while the structure of an aggregate with a variable number of components can have its composition changed during its life time.

[1] We omit cardinality when it is not important for the problem being discussed.

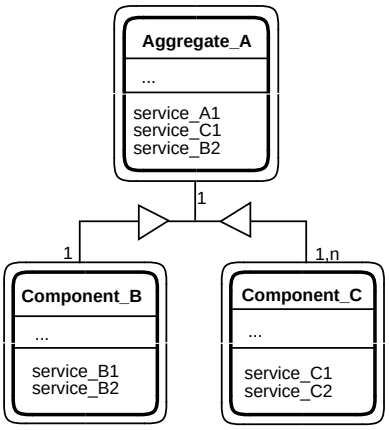

Figure 4.1: Aggregation

Aggregation with a static number of hidden components is called *disjoint* composition by Hartmann *et al.* [HJS92] and *ensemble* by Champeaux *et al.* [dCLF93]. Hartmann and his colleagues also distinguish aggregation with a static number of shared components (they call it *static non-disjoint* composition) and aggregation with a dynamic number of shared components (they call it *dynamic non-disjoint* composition).

Some authors treat other aspects of aggregation. For example, Odell classifies aggregates according to six kinds [Ode94]. Most of these kinds of composite objects do not seem to us to be useful in describing software systems.

4.5.1 Aggregation: Hidden Components

A hidden component is defined internally to its aggregate, and so it is hidden to the other objects in the system (as defined in [dCLF93, HJS92, RBP +91]). Furthermore, this object exists only while the aggregate exists. A hidden component may only interact with other components in the same aggregate or with the aggregate. The aggregate manages any interaction between such components and the rest of the model. All the services defined within hidden components which are to be offered to the outside are on offer by the aggregate. When these services are required from the aggregate by another object, the aggregate routes the message to one of its components, receives the answer, if any, and then returns the result. Figure 4.2 shows a simple object model composed of an aggregate with two hidden components and a client. We have used a modified version of the Coad and Yourdon notation. Each box in the object model represents a class template.

There is a problem in representing aggregation with hidden components. The notation used by object-oriented analysis methods, such as [CY91a, Jac92, RBP +91] is confusing, since it represents two different levels of abstraction in the same diagram. The aggregate is represented by the top box in the diagram and the components are represented by the lower boxes. However, the aggregate is really composed of everything and the lower boxes should be represented inside the top box.

In Structured Analysis, when drawing Data Flow Diagrams (DFDs), for example, we use a diagram for each level of abstraction: a DFD at level *n+1* replaces a process in the DFD at level *n*. This approach is not used to build object models. In a CASE tool,

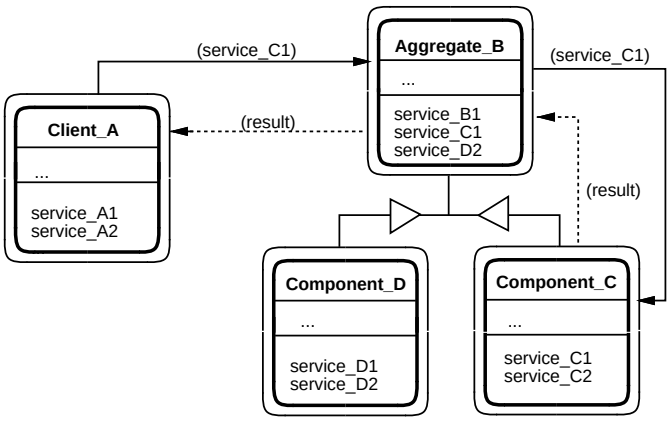

Figure 4.2: Aggregation with two hidden components

such as ObjecTool [2], subjects (or modules, or subsystems) can be expanded to show their object components, even if the resulting final diagram has a flat structure. However, an aggregate cannot be simply replaced by its components, because it is not just the collection of its components, as a subsystem is; it has extra functionality. If aggregates are useful to structure a system into levels of abstraction, we should be able to use them to build object models accordingly. Figure 4.3 represents this idea.

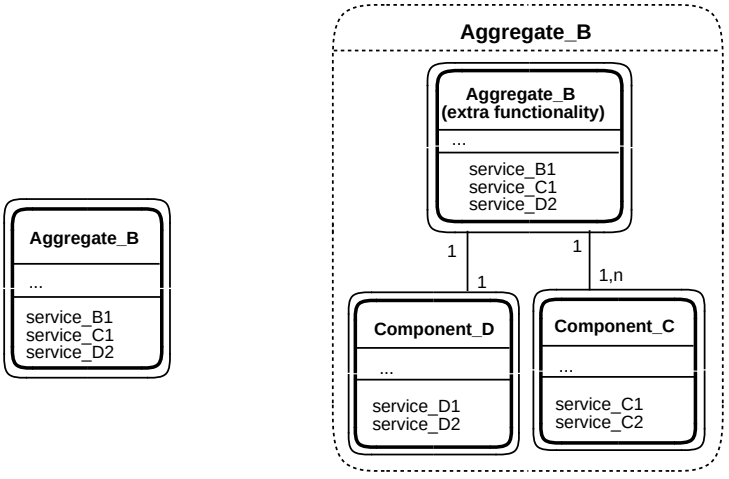

Figure 4.3: a) Higher level of abstraction: aggregate; b) Lower level of abstraction: the components and the aggregate's extra functionality

We want to use existing object-oriented CASE tools, and so we accept the standard OOA terminology. We propose an alternative interpretation of the diagram, considering, at the lower level of abstraction, the top box of the diagram to be the interface of the aggregate and the aggregate, i.e. the whole structure, as the dotted box shown in Figure 4.4.

At one level of abstraction, *Aggregate B* represents the complete structure. At a lower

[2] ObjecTool is a trademark of Object International, Inc.

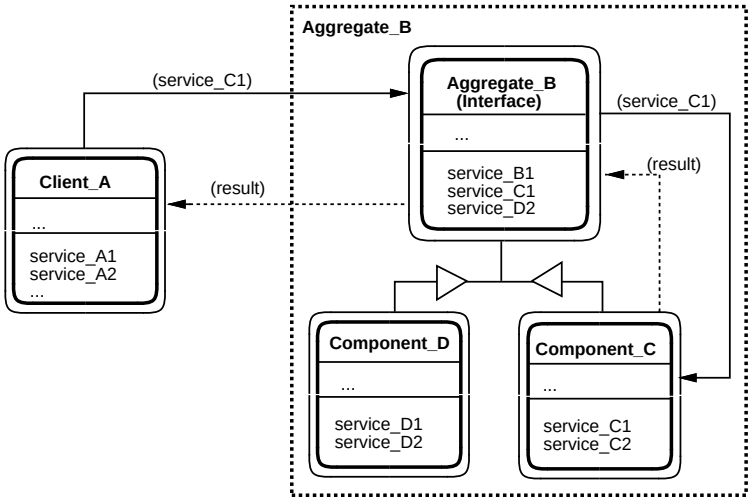

Figure 4.4: Aggregation with hidden components marked as a single class template

level of abstraction we have *Component_C*, *Component_D* and the extra functionality the aggregate offers (represented by *Aggregate_B* which is now regarded as the interface to the components). We choose to name the complete aggregate with the same name as the box at the top level of the aggregation, since they represent the same concept.

If the number of instances of each component is static, the internal structure of the aggregate never changes. If the number of object components is dynamic, the aggregate may create new objects or even remove some of the existing ones, changing its internal structure. Notice however, that in both situations, the interface of the aggregate never changes and the components only exist while the aggregate exists. The creation of new component objects is the exclusive decision of the aggregate.

4.5.2 Aggregation: Shared Components

A shared component has a life of its own, that is, it can exist independently of the aggregate. While a non-shared component is encapsulated in the aggregate and hidden from the rest of the model, a shared component is visible outside the scope of the aggregate and therefore it can communicate directly with other objects. The aggregate knows its shared components by having a specific attribute for each one. This attribute holds the object identifier of the component. Figure 4.5 shows the example given in Figure 4.2, but now *Component_C* is visible outside the aggregate.

The top box represents the whole aggregate and each lower box its components. As *Component_C* is shared, it is outside the dotted box. The services offered by the shared component can also be offered by the aggregate. It is a client's decision to choose either *Aggregate_B* or *Component_C* for communication. When an object, outside the scope of the aggregate, communicates with the aggregate, it may or may not give the identifier of the shared object involved.

There are situations which may be relevant in understanding certain problems, but which are not shown by an object model. When we talk about a shared component, we do not specify in which terms this component is shared. For example, it may be important to be able to determine whether or not a given instance of a component class is shared,

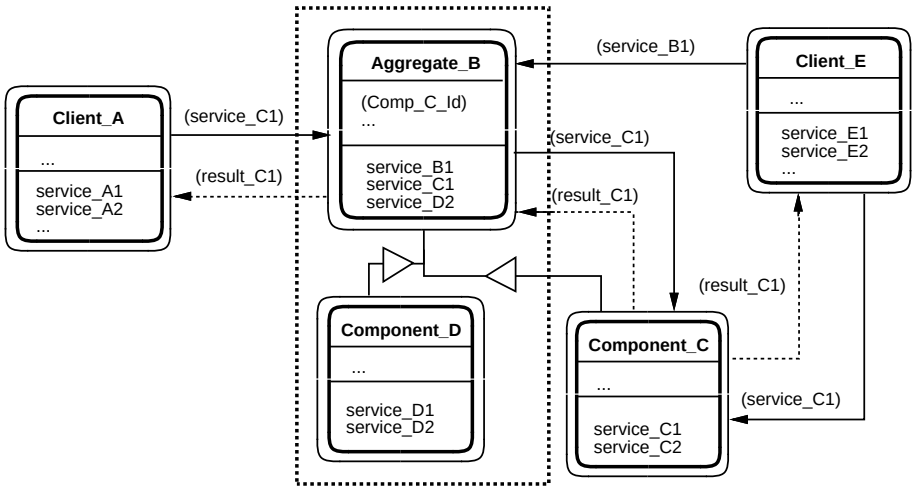

Figure 4.5: Static aggregation with one shared component and one hidden component

or if the sharing concerns different instances.

Let us consider two examples. In the first one, we define an aggregate `Car` with a component `Engine`. The engine class is shared by another aggregate, `Plane`. It seems reasonable to assume that one instance of `Engine` may not be shared by a car and a plane, at the same time. Therefore, in Figure 4.6 we have sharing of a concept, but not a sharing of objects, even if the concept of relationship is defined between instances and not between classes.

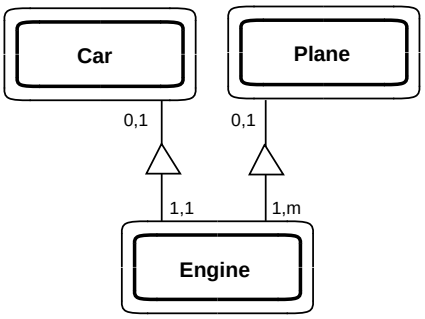

Figure 4.6: Sharing the class template, but not the objects

For the relationship between `Engine` and `Car`, `Engine` has optional cardinality, which means that an engine may or may not have a relation with an object in `Car` (similarly for the relationship between `Engine` and `Plane`). In such a situation, the object model does not distinguish between cases where an object component can belong to two aggregate objects at the same time, belong to only one of them, or to none of them.

The case in Figure 4.6 can be seen as a mixture of sharing and hiding. The component class is shared, but the component object is hidden. A component class being shared means that we can use the definition of the component to build different aggregates. (If the component class was not shared it would not be visible to other aggregates.) A component object being hidden, means that the object has to be instantiated within the

aggregate and no other objects know about it.

Now, consider the aggregates `Family` and `Research Group` sharing the same component `Person`. In this case, an instance of person can be simultaneously shared by an instance of family and by an instance of research group. As we can see in Figure 4.7, we cannot show the difference between this situation and the situation depicted in Figure 4.6 in an object model.

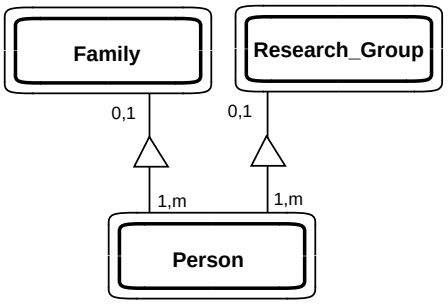

Figure 4.7: Sharing the class template and the objects

We can have static aggregation, when the number of components is constant, and dynamic aggregation, when the number of components is variable. Both cases are dealt with in a similar way to aggregation with hidden components, except that now the creation of an object component may be initiated either by the aggregate or by one of the other objects.

Hartmann *et al.* allow the creation of a dynamic aggregate with non-existing object components [HJS92]. Such components, at the moment the aggregate is defined, have an empty life cycle, but their identifiers have to be pre-determined.

According to some authors, only aggregation with hiding represents a useful mechanism [Jac92, RBP +91]. Rumbaugh *et al.*, for example, argue that aggregation with sharing should be treated as an ordinary conceptual relationship [RBP +91]. We believe that there are advantages in showing such a relationship as an aggregation as it can improve our understanding of a system and may give directions for reusability. (We justify this view in Sections 4.8.2 and 4.8.4.) While a conceptual relationship is likely to change when it is put in a different context, an aggregation may not change.

With respect to deletion, a shared component can only be removed when no other object in the system has access to it.

4.5.3 Catalog Aggregation

Some authors classify aggregation into *physical* and *catalog*, according to its cardinality [Bla93]. In physical aggregation, each component object is part of at most one aggregate object of a given class, while, in catalog aggregation, each component object may be part of many aggregate objects of the same class. That is, in a physical aggregation, each relationship from each component to the aggregate has multiplicity *one*, while in catalog aggregation, each relationship from each component to the aggregate has the multiplicity *many*. This is shown in Figure 4.8.
(The examples discussed in the previous section are physical aggregates.)

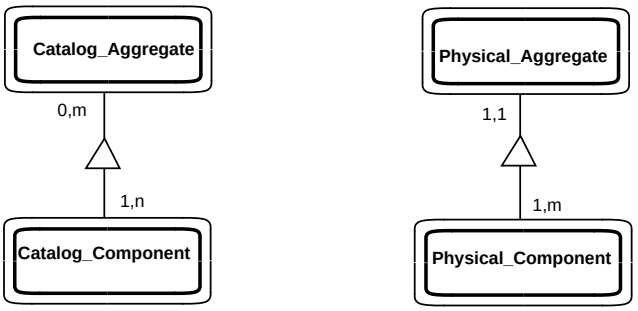

Figure 4.8: Catalog aggregation and physical aggregation

Catalog aggregation describes sharing between instances of the same class: two different instances of a given aggregate class share the same component object. In Figure 4.9 the object `Edward` is a member of the `Neural Networks` and `Artificial Intelligence` research groups.

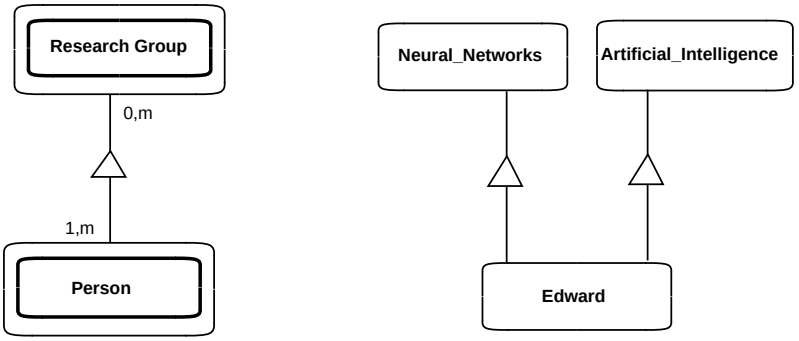

Figure 4.9: One person belongs to two research groups

When we allow catalog aggregation we lose information hiding, since an object component has to be known by two aggregate objects. Therefore, we treat this kind of aggregation as aggregation with shared component classes. Catalog aggregation is helpful when designing problems where components of the same type are interchangeable.

When discussing the design of relational data bases, Smith forbids catalog aggregation by imposing the following rule [SS77]: for a given aggregate class A with the component classes C_1, ..., C_n, two distinct real-world instances of A must not determine the same instances of C_i. That is, a given member of C_i must not be shared by two instances of A.

Kim *et al.*, when defining *dependent objects*, are implicitly defining physical aggregation [KBC+87]. Some authors classify the complex object of a physical aggregation into *aggregate*, if it has a $1:1$ relationship with each component, and *collection*, if it has a $1:N$ relationship with each component [Civ93]. A collection is useful to define homogeneous sets of objects, for example a `Football Team` is a collection of objects of the class `Footballer`. However, we do not consider this a special kind of aggregation and we use the term aggregate for both cases.

4.6 Properties of Aggregation

There are many different views of aggregation and in the previous sections we have discussed some of them. Authors seem not to agree on which specific properties aggregates should satisfy. However, we can identify one point which they all have in common: interesting aggregates have hidden components. Aggregates with hidden components are implemented using encapsulation. This permits an aggregate to be seen as a single object at one level of abstraction, and so it can be used as a structuring mechanism.

We agree that most advantages come from using aggregates with hidden components. However, instead of treating and representing aggregation-with-sharing as a regular form of conceptual relationships, we use the terminology of aggregation. By doing this, we are giving more information about the composition of an object, improving the understandability of the relationship that connects the two objects.

We propose that aggregates have the following properties:

- Aggregates may have extra functionality in addition to the functionality defined in their components;
- Object components at the same level of abstraction (within the same aggregate) can communicate with each other without having to communicate via their aggregate;
- There is a mechanism of service delegation (propagation of operations) between objects at consecutive levels of abstraction;
- An aggregate acts as the interface to, or manager of, the aggregate components. It calls services of the components, but the components do not call services of the aggregate;
- In general, aggregation is not transitive if the semantics of the relationships involved differ. Aggregation is never transitive with respect to message connections. Suppose that aggregate A has a component B and B has a component C. If A communicates with B, and B communicates with C, this does not imply that A communicates (directly) with C;
- Aggregation is antisymmetric: if B is a component of A, then A is not a component of B.

Aggregates with hidden components satisfy the following extra properties:

- An aggregate encapsulates its components;
- Each component is hidden from the rest of the world, i.e. it is not visible outside the scope of the aggregate;
- The deletion of an aggregate implies the deletion of all its components.

Aggregates with shared components satisfy the extra following properties:

- The components cannot be encapsulated as they must be visible to other objects outside the scope of the aggregate;

- If necessary, a component may know about its aggregates. This is the case for catalog aggregation, where it may be useful for a component to know to which aggregates it belongs;

- The deletion of a component is only possible if there are no other aggregates with references to it. It may be initiated by other objects in the model;

- The deletion of an aggregate does not imply the deletion of its components;

- The aggregate can exist after the deletion of its components;

- The creation of a component may be initiated by other objects in the model.

4.7 Managing Complexity

Aggregation gives a mechanism for structuring large systems. Such structuring may be accomplished either *top-down* or *bottom-up*. In a top-down approach, aggregates can show up early. As the development evolves, these complex objects are refined and their components identified. Even when we identify aggregates early, we do not have to guarantee their correct classification into hidden or shared. As we will show in Section 4.8.4, moving from one type to the other is very simple and brings no problems. In a bottom-up approach, we may start identifying the lower level objects and then we may group some of them to form aggregates.

In some situations, depending on the style of the requirements, it may be difficult to start with a top-down approach. We can identify two reasons for this. First, if the requirements are written in a functional way, when dividing a system according to its functional areas it may be that different areas describe parts of the same object. If each team takes a functional area, some of the teams may have different views of the same object. For a large project it is necessary to give names to candidate objects, without spending much time considering whether or not a given object is important. The analysts can use their knowledge about the real world. Next, we can make groupings of objects according to their role in the system and keep interactions between groupings low. Each group of objects is then given to different teams who proceed with the analysis using a mixture of top-down and bottom-up approaches.

Second, the requirements document may describe the problem in a very flat style, without hierarchical structure. It is also possible that the users describe physical details of the problem and are unable to abstract concepts. If this happens, the candidate objects we start identifying are certainly low level objects. As the development proceeds, more complex objects, such as aggregates, are identified.

Most OOA methods start by identifying objects. Depending on the style used to write the requirements document, we can identify aggregates sooner rather than later. We believe that in a large project the objects we start identifying are a mixture of complex and simple objects. We can identify complex objects in two ways: by analysing relationships and similarities between objects (bottom-up composition) and, while describing an object we may identify it as being a complex object (top-down decomposition).

4.8 Modelling Aggregation in LOTOS

We have shown in Chapter 3 how to model objects and classes in LOTOS. An object is normally modelled by a process and one or more ADTs, but when an object in the object model only plays the role of attribute of another object, it is modelled by a single ADT.

To be useful as an abstraction, an aggregate must play an important role in the system and so it is modelled as a process and one or more ADTs. If the object only plays the role of an attribute, although an attribute may have internal structure, we do not regard it as an aggregate. For example, although we can consider the attribute `Address` to be composed of `House_Number`, `Street Name`, `Post Code`, `City Name` and `Country Name`, `Address` is modelled as an ADT which is a combination of sorts (one sort for each component). Also, if the components of an object only play the role of attributes, the object is modelled as an ordinary object, not as an aggregate.

This section is concerned with showing how to model aggregates in LOTOS. As many authors suggest, we model aggregation with shared components as conceptual relationships. The aggregate and each component are defined by separate processes, and the aggregation (the relationship) is modelled in the same way as for conceptual relationships. Unless something different is said in the requirements, the aggregate knows its components, but the components do not know about the aggregate.

An aggregate with hidden components is modelled as a process which encapsulates a process for each of its components together with a process manager, or interface. As a rule, we give the same name to both the process representing the aggregate and the process representing the interface. The following subsections discuss this in detail. A hybrid aggregate has hidden and shared components. The hidden components are modelled as processes embedded in the process defining the aggregate, while the shared components are modelled as separate processes. A reference to each separate process is modelled as an ADT and given as a parameter of the aggregate's process.

While a simple object may be regarded as a sequential machine, an aggregate has the implicit connotation of having internal parallelism [dCLF93].

Having a static or a dynamic number of object components does not complicate the problem, since we can easily define object generators. If we are dealing with aggregation with hidden components, the object generators for each component are defined inside the aggregate and therefore they are not visible from the outside. This allows the same set of identifiers to be used by different object generators defined inside different aggregates. The advantage of this is its simplicity. However, each component is then only uniquely identified when the aggregate identifier is given together with the component identifier. We do not see this as a problem, since the components are not visible to the other objects in the system and so the aggregate identifier must always be used to access them.

To demonstrate how aggregates can be modelled in LOTOS, we use a simple video player with four functions: load a tape, play a tape, stop playing and eject a tape. The video has two components: a motor and an eject mechanism. Its behaviour is given by the finite state automaton depicted in Figure 4.10.

4.8.1 Aggregation with Hiding

If the number of components is constant, we can define the internal composition of the aggregate at specification time. Creation or deletion of the aggregate implies the creation

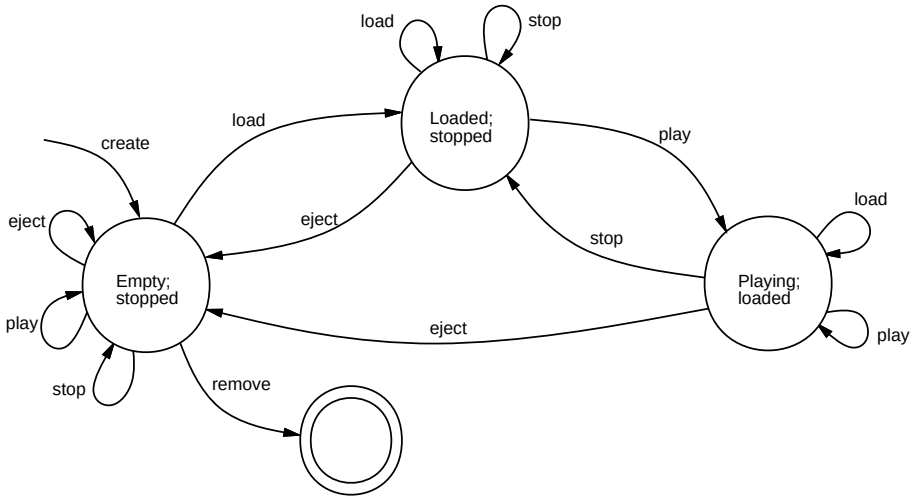

Figure 4.10: Behaviour of the video player example defined as a finite state automaton

or deletion of its components. Therefore, the deletion of a component is impossible without the deletion of the aggregate.

Let us suppose that the video player has one motor and one eject mechanism. Figure 4.11 shows the class templates for `Video`, each one with its attributes and services. `State_V` can take the values `empty`, `loaded` and `playing`; `State M_`can take the values `stopped` and `run`; finally, `State E` can take the values `empty` and `loaded`.

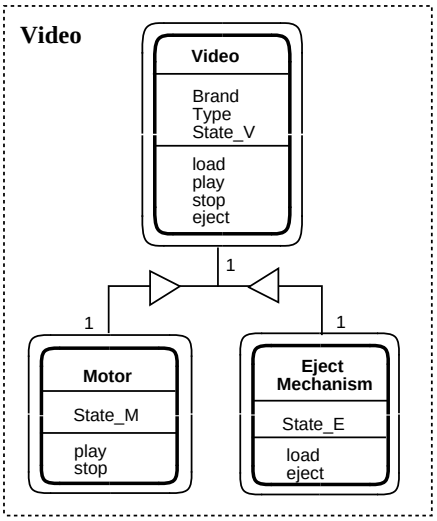

Figure 4.11: Video player aggregate

As the components are not shared, we draw a dotted box (structure) around the aggregate. The semantics of this is that the class components `Motor` and `Eject Mechanism` are encapsulated within the aggregate `Video`, and so they are not visible outside `Video`. In Chapter 6 we show, by following an algorithm, how to build an Object Communication Diagram (OCD) as depicted in Figure 4.12. (In this simple example, the rest of the system

is the interface scenario.)

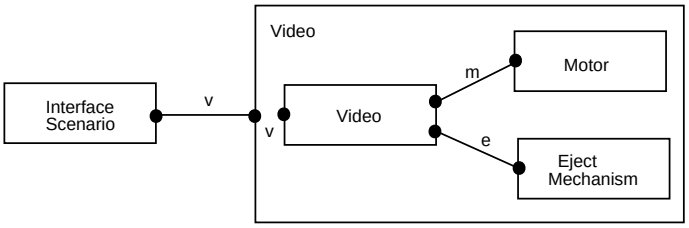

Figure 4.12: Object Communication Diagram

In LOTOS, the corresponding top level behaviour expression takes the form:

```
( Video[v](Make_Video(id1 of Video_Id, empty of State_V, brand_video,
                      type_video, id1 of Motor_Id, id1 of Eject_Mechanism_Id))
|[v]|
  Interface_Scenario[v]
)
where
  process Video[v](this_video: Video_State): noexit :=
    hide m, e in
    ( Video[v, m, e](this_video)
    |[m,e]|
      ( Motor[m](Make_Motor(Get_Motor_Id(this_video), stopped of State_M))
      |||
        Eject_Mechanism[e]
          (Make_Eject_Mechanism(Get_Eject_Mechanism_Id(this_video),
                                empty of State_E))
      )
    )
  where
    process Video[v, m, e]      ... endproc (* Video *)
    process Motor[m]            ... endproc (* Motor *)
    process Eject_Mechanism[e] ... endproc (* Eject_Mechanism *)
  endproc (* Video *)
```

As we discussed before, we name the complete aggregate with the same name as the box at the top level of the aggregation. This is shown in LOTOS by having two processes with the same name, one encapsulating the other. At one level of abstraction the outside process represents the aggregate (i.e. the box at the top level of the hierarchy) while at a lower level of abstraction the inner process represents the box at the top level of the hierarchy which is now regarded as the interface to the component objects.

As gates `m` and `e` are hidden in the external process `Video`, `Motor` and `Eject Mechanism` are encapsulated within `Video` and defined after the keyword `where`. The two components are therefore hidden from `Interface Scenario`. In this example we are only creating one instance of each class template `Motor` and `Eject Mechanism`, but we could create others by having more process instantiations in the behaviour expression. The operations `Get_Motor_Id` and `Get_Eject_Mechanism_Id` are defined in the ADT that defines the sort `Video_State`, as follows:

```
type Video_Type is Video_Id_Set_Type, State_V_Type, Brand_V_Type,
                   Type_V_Type, Motor_Id_Set_Type, Eject_Mechanism_Id_Set_Type
  sorts Video_State
  opns
    Make_Video             : Video_Id, State_V, Brand_V, Type_V, Motor_Id,
                             Eject_Mechanism_Id   -> Video_State
    Change_State           : Video_State, State_V -> Video_State
    Get_Video_Id           : Video_State          -> Video_Id
    Get_Motor_Id           : Video_State          -> Motor_Id
    Get_Eject_Mechanism_Id : Video_State          -> Eject_Mechanism_Id
    ...
eqns forall v: Video_State, b: Brand_V, t: Type_V, n: Video_Id, m: Motor_Id,
            e: Eject_Mechanism_Id, s, s1: State_V
  ofsort Video_Id
      Get_Video_Id(Make_Video(n, s, b, t, m, e)) = n;
      Get_Video_Id(Change_State(v, s))           = Get_Video_Id(v);
  ofsort Motor_Id
      Get_Motor_Id(Make_Video(n, s, b, t, m, e)) = m;
      Get_Motor_Id(Change_State(v, s))           = Get_Motor_Id(v);
  ofsort Eject_Mechanism_Id
      Get_Eject_Mechanism_Id(Make_Video(n, s, b, t, m, e)) = e;
      Get_Eject_Mechanism_Id(Change_State(v, s))
        = Get_Eject_Mechanism_Id(v);
  ofsort ...
endtype
```

In LOTOS, defining an aggregate with a dynamic number of components is not difficult, since we have the facility of defining object generators. An object generator allows us to create multiple instances of objects. Supposing that the number of components was dynamic, the external process `Video` would be defined as:

```
process Video[v](this_video: Video_State): noexit :=
  hide m, e in
    ( Video[v, m, e](this_video)
    |[m, e]|
      ( Motors[m]({} of Motor_Id_Set)
      |||
        Eject_Mechanisms[e]({} of Eject_Mechanism_Id_Set)
    ) )
where
   ...
endproc (* Video *)
```

in which `Motors` and `Eject_Mechanisms` are object generators, each one initialised with an empty set of identifiers. As the components are hidden, their process definitions are in the scope of the external process `Video`, after the keyword `where`. As an example, let us consider the object generator `Motors`:

```
process Motors[c](mts: Motor_Id_Set): noexit :=
  c !create ?id: Motor_Id [id notin mts];
  ( Motor[c](Make_Motor(id of Motor_Id, stopped of State_M))
  |||
```

```
      Motors[c](Insert(id, mts))
    )
  where
    process Motor[c](this_motor: Motor_State): noexit :=
      ( [Get_State(this_motor) eq stopped] ->
          ( c !play !Get_Motor_Id(this_motor);
            exit(Change_State(this_motor, run))
          )
      []
        ...
    endproc (* Motor *)
  endproc (* Motors *)
```

`Motors` holds the set of identifiers already created. (Notice that for simplicity we are using the same gate `c` to create a motor and to operate the motor, but we could use two different gates.)

When `Interface_Scenario` requires a service, `Video` routes the request to the right component and then returns the result, if any. When more than one object component is ready to synchronize, one component will be chosen non-deterministically. The inner `Video` process, i.e. the process defining the interface of the aggregate, can be defined as:

```
process Video[v, m, e](this_video: Video_State): noexit :=
  ( hide create_motor, create_eject in
     [Get_State(this_video) eq loaded] ->
       ( v !play !Get_Video_Id(this_video);
         m !play ?m1: Motor_Id [m1 IsIn Get_Motor_Id_set(this_video)];
         exit(Change_State(this_video, playing))
       []
         ...
       )
  []
    ...
  []
    create_motor;
    m !create ?idm: Motor_Id;
    exit(Add_Motor(this_video, idm))
  []
    create_eject;
    e !create ?ide: Eject_Mechanism_Id;
    exit(Add_Eject_Mechanism(this_video, ide))
  ) >> accept upd_video: Video_State in Video[v, m, e](upd_video)
endproc (* Video *)
```

where `create_motor` and `create_eject` are internal events. The behaviour expression:

```
m !play ?m1: Motor_Id [m1 IsIn Get_Motor_Id_set(this_video)];
```

may synchronize with any motor which is in the right state and whose identifier is known by the aggregate.

The creation of new motors is defined in the body of the inner process `Video`, by using the internal event `create_motor` and then synchronizing on event:

```
m !create ?idm: Motor_Id;
```

with the corresponding event defined in the object generator. The `Eject Mechanism` is dealt with in a similar way.

The operation `Add_Motor` and `Add_Eject_Mechanism` are defined in the ADT `Video_Type`. This ADT has to be changed to support sets of motors and sets of eject mechanisms.

When other objects in the system know about the existence of the components, they can initiate the creation of a new component. However, the components are not visible, and so it is always the aggregate's responsibility to select the right component and to create new ones, by using the object generators.

As a result of having encapsulated components, another instance of `Video` can use the same component identifiers. This means that the identifier of a component must be combined with the identifier of the aggregate to give the full object component identifier.

Although a dynamic number of hidden components can be specified in LOTOS, we believe that it is not a common case. One situation is when a component breaks down. If a component is not responding to the services required, the aggregate can substitute it with a new component. The AT&T ESS5 switch has 'auditors' which go around checking invariants in software modules which have a certain functionality. If a module does not satisfy the invariant, the auditor shuts them down and reinitialises them, or replaces them with other instances [Hoa94].

4.8.2 Aggregation with Sharing

A shared component exists independently of the aggregate. It is modelled in the usual way, with a class template and perhaps an object generator, but outside the process defining the aggregate. The aggregation relationship will then be modelled as a conceptual relationship in the ADT that defines the state information. In general, the aggregate has a reference to the shared component, while the shared component only has a reference to the aggregate if it is explicitly required.

As the components are shared, the deletion of the aggregate does not imply the deletion of its components, as happened in the previous section. However, the deletion of a component may imply the deletion of the aggregate.

Let us recall the video player example, where we have two object components, one for each component class. Figure 4.13 shows the OCD supposing that `Eject Mechanism` and `Motor` can be directly accessed by `Interface Scenario`.

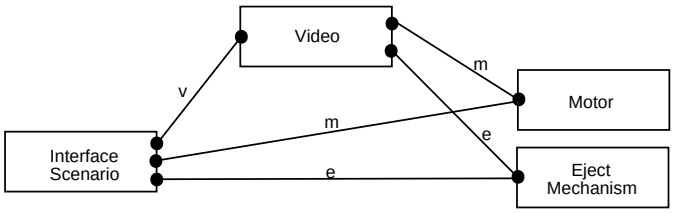

Figure 4.13: Object Communication Diagram

In LOTOS, the top level behaviour expression takes the form:

```
( Interface_Scenario[v, m, e]
```

```
|[v]|
  Video[v, m, e](Make_Video(id1 of Video_Id, empty of State_V, brand_video,
                    type_video, id1 of Motor_Id, id1 of Eject_Mechanism_Id))
)
|[m, e]|
  ( Motor[m](Make_Motor(id1 of Motor_Id, stopped of State_M))
  |||
    Eject_Mechanism[e]
       (Make_Eject_Mechanism(id1 of Eject_Mechanism_Id, empty of State_E))
  )
where
 ...
```

The difference between this situation and aggregation with hidden components is that the communication gates `m` and `e` are visible from the `Interface_Scenario`. Process `Video` corresponds to the interface in the case of aggregation with hiding, although here we want to regard it as the whole aggregate. The ADT that specifies the video state information is unchanged. The information about the relationship between `Video` and `Motor` and between `Video` and `Eject Mechanism` is given by the parameters of `Make Video`. Instead of modelling these relationships in the ADT, we could model them as extra parameters of the process `Video`, as we do for normal conceptual relationships. However, we prefer the first option.

If we had a dynamic number of object components, the parameters of `Make Video` would be sets of identifiers, instead of single identifiers, and the LOTOS top level behaviour expression would instantiate an object generator for each component, instead of instantiating each component.

We can also model catalog aggregation. For example, suppose that two videos could share a motor and an eject mechanism. This can be represented as:

```
( Interface_Scenario[v, m, e]
|[v]|
  ( Video[v, m, e](Make_Video(id1 of Video_Id, empty of State_V,
                    brand_video, type_video, id1 of Motor_Id,
                    id1 of Eject_Mechanism_Id))
  |||
    Video[v, m, e](Make_Video(id2 of Video_Id, empty of State_V,
                               id1 of Motor_Id, id1 of Eject_Mechanism_Id))
  )
)
|[m, e]|
  ( Motor[m](Make_Motor(id1 of Motor_Id, stopped of State_M))
  |||
    Eject_Mechanism[e]
       (Make_Eject_Mechanism(id1 of Eject_Mechanism_Id, empty of State_E))
  )
where
 ...
```

When modelling catalog aggregation we may want to change the ADT of the component to deal with an extra attribute which gives the set of aggregates in which it takes part.

In the example given in Section 4.5.3, it is desirable that a person knows about his or her research groups.

While in the case of static aggregation with hiding the aggregate had sole responsibility for creating its components at specification time, in the case of dynamic aggregation the creation and deletion of an object component may be initiated by other objects in the system. Also, if the object requiring the service knows the identifier of the object component which will be involved in the operation, we can use that information when asking the service to `Video`, instead of using value generation:

```
process Video[v, m, e](this_video: Video_State): noexit :=
  ( [Get_State(this_video) eq loaded] ->
    ( v !play !Get_Video_Id(this_video) ?m1: Motor_Id
         [m1 IsIn Get_Motor_Id_set(this_video)];
      m !play !m1;
      exit(Change_State(this_video, playing))
    []
      ...
    )
  []
   ...
endproc (* Video *)
```

4.8.3 Sharing Concepts but not Objects

In Sections 4.5.2 and 4.5.3 we discussed some different views of sharing. When the component class is shared, the object components may or may not be. The LOTOS behaviour expressions in Section 4.8.2 give us both shared component classes and shared object components. A shared component class and non-shared object component can be obtained by proceeding as we did for hidden components, but now the component class templates are defined outside the scope of the external `Video` process while they are instantiated within its scope:

```
process Video[v](this_video: Video_State): noexit :=
  hide m, e in
  ( Video[v, m, e](this_video)
  |[m,e]|
    ( Motor[m](Make_Motor(Get_Motor_Id(this_video), stopped of State_M))
    |||
      Eject_Mechanism[e]
         (Make_Eject_Mechanism(Get_Eject_Mechanism_Id(this_video),
                               empty of State_E))
    )
  )
where
  process Video[v, m, e] ... endproc (* Video *)
endproc (* Video *)

process Motor[g] ... endproc (* Motor *)
process Eject_Mechanism[g] ... endproc (* Eject_Mechanism *)
```

4.8.4 Hiding and Sharing: Moving Around

Having modelled aggregates with hidden components and aggregates with shared components in LOTOS, let us discuss the changes necessary to transform one into the other. The basic difference in modelling these two kinds of aggregates is concerned with encapsulation and information hiding. While hidden components are encapsulated by the aggregate and hidden from the other objects in the model, shared components are modelled as separate processes which are visible from outside the aggregate.

As we have discussed in Section 4.5.1, in order to encapsulate a hidden component into its aggregate, we create a higher level structure which we name with the same name as the class template at the top of the aggregation hierarchy in the object model.

To transform an aggregate with hidden components into an aggregate with shared components, we follow the two steps:

1. Replace the process that defines the higher level structure, i.e. the outside process, by its behaviour expression which joins the inner process with the components. The inner process now plays the role of the higher level structure.

2. Make all the gates visible outside the aggregate's scope, removing the `hide` operator, and add those gates to any process instances which need them.

The opposite, i.e. transforming an aggregate with shared components into an aggregate with hidden components, can be accomplished in two steps:

1. Encapsulate the aggregate and components processes within an extra process. Name this process with the same name as the previous aggregate. (Now, the outside process represents the aggregate and the inner process represents the interface of the whole structure with the other objects in the model.)

2. Hide, in the encapsulating process, the gates which are used to communicate with the aggregate components.

The procedures above can both be applied when dealing with object generators.

4.9 Conclusions

Aggregation is a useful concept which can be used to control the size and complexity of a large system. Aggregation with hidden components mainly uses the concepts of abstraction, encapsulation and information hiding. This helps in providing a top-down approach which aids the software engineer developing the system and, at the same time, guides the reader to understand the system.

Aggregation with shared components does not bring as many advantages and many authors advocate that it should be treated as an ordinary conceptual relationship. However, we believe that there are advantages in showing it in an object model. It will give hints about the structure of a system, helping us to understand it, and it can also give directions for reusability. While a conceptual relationship is more likely to change when the system is put in a different context, an aggregation may not change and so we can see it as reusable in other contexts. That is why we propose modelling it within the ADT, instead

of modelling it as an extra argument in the process template, as we do with conceptual relationships.

LOTOS can model aggregation with shared and with hidden components. Encapsulation is dealt with by changing the scope of processes and using the `hide` operator. By defining sets and object generators, LOTOS also deals well with the cases of a static and a dynamic number of object components.

Chapter 5

Formalising Object-Oriented Analysis With LOTOS

5.1 Introduction

The purpose of this chapter is to specify the components of an object model and their relations in such a way as to illuminate the formal syntactic transformations used in OOA methods, more precisely, in the ROOA method. One goal is to provide a simple-as-possible denotation of each concept so that practitioners of object-oriented analysis can get on with the hard job of building models without, say, having to learn category theory, universal algebra or fixpoint theory. The actual formalisation of the concepts is thus of secondary importance, accomplished where possible simply by listing components as tuples, and functions and relations by giving their denotational type (i.e. the domain and range sets). Nevertheless, there are technical problems arising with even so straightforward an approach. Our main concern is to identify and solve these problems. The main problem solved here is: how the behaviour of objects (in the system) is derived from the expressions of generic behaviour contained in the class templates in the model.

In this chapter we will only handle the concepts we believe are the more important and the basic ones. The theory we present can be used to extend the definitions over the rest of the OOA concepts.

We assume some familiarity with the Z notation [Spi89].

5.2 The Reasons For this Work

The first reason is precision, i.e. the formalisation of object modelling concepts. We motivate the approach taken in ROOA and its semantics by contrasting the goals of ROOA with those of other work.

Another reason is enhanced expressive power. Object models suggested in the OOA literature are rendered less expressive by their dependence on particularly simple behavioural models (e.g. finite automata [CY91a, RBP $^{+}$91]) and particular notions of atomic action. The semantics of LOTOS, used for ROOA, allows much richer expression of behaviour than these proposed models.

Other approaches to giving formal structure to object-oriented concepts have started with objects as fundamental and derived other concepts from these. They are mainly

concerned with object-oriented *programming*, or with object-oriented *design*, not with OOA. Examples include a formalisation [BMS93] using Lamport's TLA logic [Lam94], and formalisations using universal algebra and category theory [EGS93, EGS91, ESS89]. These approaches are as powerful as LOTOS for describing the behaviour of objects. However, the existing object-oriented analysis methods give primacy to class templates and their meaning, and objects are simply not around. (Objects only appear during the execution of an executable model, such as the one created by ROOA, but not in the definition of that model.) Hence those approaches do not appear to explain the concepts of OOA in practical methods such as OMT as ROOA does, in a way understandable by practitioners without advanced mathematical experience.

Furthermore, those other approaches do not make a clear distinction between static structure and behaviour, in the way emphasised by methods such as OMT [RBP $^{+}$91] or some approaches to Open Distributed Processing [Got93]. Recovery from failure (e.g. [LS83]) cannot be properly solved in the analysis stage. The philosophy of OOA suggests that they should be addressed at the design stage, using methodologies developed for this purpose. According to this philosophy, such classes of problems should be separated from the architecture of the system as expressed by the specification text, as in ROOA.

Other approaches to the development of formal system specifications in an object-oriented manner may be found in the paper by Cusack and Lai [CL91], in the work proposed by Jones [Jon93] and in the work proposed by Laorakpong and Saeki [LS93]. These approaches introduce new methodology in order to obtain formal specifications, and it is not clear that they build on practical approaches such as [Jac92, RBP $^{+}$91], as in ROOA. The goal of Cusack and Lai is to integrate object-oriented concepts into LOTOS specifications (rather than the other way around, as in ROOA), and "the immediate area of application ... is the development of international standards for ... Open Distributed Processing (ODP) systems". A similar goal with respect to VDM-style development underlies the work of Laorakpong and Saeki. Finally, Jones is concerned with a formal notation for object-oriented programming and design, not with OOA. Because of their goals, these approaches focus on the specification of objects rather than the concepts of OOA, and do not strictly adhere to the division between analysis and design recommended by [CY91a, Jac92, RBP $^{+}$91, SM89] and employed in ROOA.

5.3 Basic Concepts: Set of Values and Variables

First we define the set *set of values* of all possible sets of values of data types. We also define an infinite set of *variables*. Specification languages have various ways to handle data typing, some extensional (data types are identical if their sets of possible values are identical, no matter what the names) or intensional (not extensional). We only need to use data values (i.e. extensional data typing) for this work, but a language such as LOTOS does not have fully extensional data typing. We must therefore be flexible enough in what we do to cohere with the data typing principles we might encounter in this and other languages. We define $\mathcal{V}$ as the set containing all data values, and *data types* as the set of data types (whatever they may be). We define a separate naming function *extension* which associates a data type with a set of values, and do not say (because we do not need to) what the properties of this function are, or how identities between data types are handled. Our development ensures that the sets *class templates*, *objects*, *attributes*,

services, *data_types*, *message_types* are all disjoint. We further require that the two sets

$$set\_of\_values = \{v \mid v \subseteq \mathcal{V}\} = \mathbb{P}\,\mathcal{V}$$
$$variables = \{x_i \mid i \in \mathbb{N}\}$$

are disjoint from each other and from everything else in sight. We define a naming relation

$$named\_by : STRING \times (variables \cup data\_types \cup class\_templates \cup objects \cup \cup attributes \cup services \cup message\_types \cup object\_generators)$$

which assigns names to everything in sight. We also have a mapping

$$extension : data\_types \rightarrow set\_of\_values$$

which gives the extension of each data type. Finally the set

$$variable\_type\_pairs = \{\langle x, d\rangle \mid x \in variables, d \in data\_types\}$$

allows binding of variables in a particular context with data types.

The justification for these definitions is as follows. We accumulate all possible set-of-values values in $\mathcal{V}$, and therefore a specific data type will have an extension which is a specific subset of the values of $\mathcal{V}$. We need an infinite set of variables for standard syntactic reasons. Variables in a particular use are generally bound to data types: we want to be able to select a variable x of specific type *real*, say, in a given application. Therefore we need a set of variable-to-data-type bindings.

5.4 Defining the Concepts of Object-Oriented Models

We define the various concepts *class template*, *attribute* and *service* in the static structure. We may define from these concepts the derivative notions of *object* and *class* which are not in the object model, but in the implemented system. An object is an instance of a class template, which is to say that it has the attributes specified in the template, the services provided therein, and the behaviour defined therein. The notion of embodiment is thus quite simple: since everything is defined in the template except the identity of particular objects (although their sort is defined), an object is defined by assigning an identifier, and assigning behaviour by instantiating a free variable in the class template to the object identifier. A class is defined as the set of all objects that may be instantiated from a given class template. As we show in Chapter 3, in ROOA we define the instantiation of objects from class templates by means of an artifact we call an *object generator*.

Suppose the item *item* is the name of an element of a tuple (or sequence or record) defining *concept*. We shall in general write *concept.item* to refer to the value of this *item* in *concept*, as one normally does for records.

5.4.1 Class Template

A class template is defined by a *name*, a finite set of attributes Σ_A, a finite set of services Σ_S and a behaviour description $\mathcal{B}$ which apply to all objects instantiating this template. Thus, we define:

$$class\_template = \langle name, \Sigma_A, obj\_id, \Sigma_S, \mathcal{I}, \mathcal{B}\rangle$$

where $name \in dom(named\_by \rhd class\_templates)$ is the name of the class template [1]. There is a distinguished attribute $obj\_id \in \Sigma_A$, known as the object identifier attribute. Values of obj_id are the possible identifiers of the different objects instantiated from $\mathcal{B}$. The function $\mathcal{I} : \mathcal{S}_{c_t} \rightarrow \mathbb{P}\,\Sigma_S$, describes the services that are available for each given state of an instantiated object, i.e. the interface of an object at a given state. This function is explained in Section 5.4.3. $\mathcal{B}$ describes the typical behaviour of objects instantiated from the class template, and is defined by means of LOTOS constructs. We thus commit ourselves here to a specific form for $\mathcal{B}$. It is, however, important that no matter what language $\mathcal{B}$ is formulated in, it has certain properties which may be expressed using concepts from the syntax of logic. (The function of $\mathcal{B}$ is explained in Section 5.4.2.) The purpose of this requirement is that a class template c_t defines the behaviour of arbitrary objects instantiating c_t, and mentions no specific object by name. It may also refer to objects instantiating other templates, including the services they offer, again without mentioning specific objects by name. This is accomplished in logical syntax simply by using free variables.

The syntactic details of how this is accomplished in particular behaviour description languages such as LOTOS, SDL, Manna-Pnueli temporal logic, or TLA, are dependent on these particular languages. Therefore we explain such syntactic restrictions here generically, but illustrate with the ROOA derivation in LOTOS. Individual languages will require that the restrictions are translated into the syntactic restrictions that make sense for these languages.

5.4.2 The Generic Behaviour Description $c_t.\mathcal{B}$

The behavioural description $c_t.\mathcal{B}$ contained in a definition of c_t may have different forms, depending on the description language chosen. In ROOA, it is part of a LOTOS specification, usually a LOTOS process definition. But a semantic explanation of $c_t.\mathcal{B}$ should also explain its function if expressed in another description method such as SDL or TLA. We explain it here and in Section 5.5.3.

The behavioural description $c_t.\mathcal{B}$ is *generic*, in that it should provide a schema for describing the behaviour of an arbitrary object instantiated from c_t. We describe features we require of $c_t.\mathcal{B}$ so that it may be transformed into a description of the behaviour of an individual object instantiated from c_t. The procedure for transformation itself is described in Section 5.5.3.

We describe the structure of $c_t.\mathcal{B}$ in general syntactic terms taken from logic, using the notions of (logical) *individual variable* [2], and *substitution*. We need to interpret these notions in a given target language (LOTOS, SDL, TLA, *etc.*). We illustrate with a LOTOS example in ROOA.

Since $c_t.\mathcal{B}$ is intended to be a generic behaviour expression, it must contain terms which are not constant, but which take different values when used to describe the behaviour of a particular object. (In logic, these terms are said to *range over* their collection of values.) There is no bound on the number of objects instantiated in a system from the class templates in the object model. We require a set Y of individual variables ranging over

[1] For definition of the domain and range restriction operators $\lhd$ and $\rhd$, as well as the domain and range selectors *dom* and *ran*, see [Spi89, pages 96, 98]

[2] This notion should of course be distinguished from the notion of program variable, or *variable* as used in LOTOS.

values of the attribute *obj_id* (that is, over the set $dom(named\_by \rhd objects)$). We also use the set of terms C referring to the class templates. There is a fixed set of class templates in the model, namely, the set $ran(named\_by \rhd class\_templates)$, and the set of their names is $C = dom(named\_by \rhd class\_templates)$.

When a variable $y \in Y$ is used in the behaviour expression $c_t.\mathcal{B}$ to refer generically to an object, it must range over objects instantiated from a particular class template c. We denote this binding by $c.y$. We also require a distinguished individual variable ι, used in $c_t.\mathcal{B}$ in the form $c_t.\iota$. When $c_t.\mathcal{B}$ is used to describe the behaviour of a particular object ob the value of whose *obj_id* is *oid*, the term $c_t.\iota$ will be replaced by *oid* (see Section 5.5.3). Thus $c_t.\iota$ is used to refer to the *'currently instantiated object'*. This corresponds to the concept called *'self'* in Smalltalk, or *'this'* in C++. See Section 5.5.1 for definition of the concept *object*, and Section 5.5.3 for an explanation of how the behaviour of the object ob is obtained from the expression $c_t.\mathcal{B}$.

When $c_t.\mathcal{B}$ mentions a service of an object instantiated from a different class template c, we write the occurrence as $c.y.service.name$, where $c \in C$, $y \in Y$, and $service \in c.\Sigma_S$ (the set of services offered by template c). Similarly, when an attribute of an object instantiated from c_t is referred to (only attributes of objects instantiating the current class template may be referred to — all values of attributes of objects instantiating other templates must be accessed by services offered by those objects) then a term of the form $c_t.y.attribute.name$ is used.

In order to explain how the generic behaviour expression $c_t.\mathcal{B}$ is used to generate an expression describing the behaviour of a particular object in the system with identifier *oid*, we must identify the terms of the form $c_t.\iota$, $c.y$, $c.y.service.name$ and $c_t.y.attribute.name$ and all occurrences of them in $c_t.\mathcal{B}$.

5.4.3 The Visibility Function $\mathcal{I}$

Objects have 'state' [RBP+91]. We define the *value* of an object at any point to be the collection of values of the attributes of the object. We identify the *state* of the object at a given time with its value. In any particular state, not all of the services may be offered by the object. $\mathcal{I}$ is the *visibility* function, that for each state of an object tells which services are visible in that state. In the banking system problem we have been using through the chapters of this thesis, we could impose, for example, that in an `Account` object, a `withdraw` service cannot be offered if the value of `Balance` is 0 (see Section 5.4.4). With each state of the object we may associate the set of services which are offered in that state. However, which services are offered in a particular state is (a) dependent on properties of the state of that object alone; and (b) invariant over all objects instantiating the same template. For example, the fact that `withdraw` is not offered in a state in which `Balance` is 0 is a generic constraint on all objects of type `Account`. Thus, visibility is statically determined: the collection of all such constraints, the *visibility* function, is a feature of the class template. Where $\mathcal{S}_{c_t}$ is the set of states (defined below), $\mathcal{I}$ has the type $\mathcal{I} : \mathcal{S}_{c_t} \rightarrow \mathbb{P}\,\Sigma_S$.

Since a state of an object is its value, the set of attribute-value pairs of the object at a particular time, we require that the set of states $\mathcal{S}_{c_t}$ is the collection of all possible combinations of values of all attributes. But not all of these sets of attribute-value pairs may be attained by some object in some run of some system, in other words, these potential states may not be *reachable*. However, it is often combinatorially intractable to determine which

states are reachable [RS88]. Furthermore, it is evident from reachability analyses that the distinction between those states which are reachable or not is properly a logical consequence of the behavioural description of the system, and thus of logical properties of the behavioural descriptions $c.\mathcal{B}$ for all class templates c. Therefore the distinction between reachable and non-reachable states should not be made in the denotational description of the class template. Thus, we may define $\mathcal{S}_{c_t}$ to be the collection of all assignments of values of the right type to attributes in Σ_A, where $a.value\_set$ is the value domain of the attribute (see Section 5.4.5):

$$\mathcal{S}_{c_t} = \{f \mid f : \Sigma_A \rightarrow \mathcal{V} \wedge \forall\, a \in \Sigma_A \bullet f(a) \in a.value\_set\}$$

where $value\_set \in data\_types$.

5.4.4 Class Template in LOTOS

Consider the banking system defined in Section 3.3.1. A possible definition of the class template `Account` is [3]:

```
process Account[a](this_account: State_Account) : noexit :=
 ( a !deposit !Get_Account_Number(this_account) ?m: Money;
   exit(Credit_Account(this_account, m))
 []
   a !get_balance !Get_Account_Number(this_account)
     !Get_Balance(this_account);
   exit(this_account)
 []
   ( choice zero_balance: Bool []
     [not(zero_balance)] ->
     ( a !withdraw !Get_Account_Number(this_account) ?m: Money;
       ( choice enough_money: Bool []
         [enough_money] ->
           a !rtn_withdraw !Get_Account_Number(this_account) !true;
           exit(Debit_Account(this_account, m))
       []
         [not (enough_money)] ->
           a !rtn_withdraw !Get_Account_Number(this_account) !false;
           exit(this_account)
       )
     )
     [zero_balance] -> exit(this_account)
   )
 ) >> accept update_account: State_Account in Account[a](update_account)
endproc
```

We explain how to identify the various syntactic components of the object definition in the LOTOS expression informally. The name of the class template is `Account`; the set of attributes Σ_A, including the obj_id, are given in the ADT where the sort `State Account` is specified; the set Σ_S of services are defined as the behaviour expressions of the choice operator '[]'. The signatures of the services are the first action prefixes of those behaviour expressions together with any corresponding return events. Specifically, the signatures of the three services offered by `Account` are:

[3] For simplicity, we are ignoring the fact that `Account` is an abstract superclass.

```
a !deposit !Get_Account_Number(this_account) ?m: Money;
a !get_balance !Get_Account_Number(this_account) !Get_Balance(this_account);
```

for the services `deposit` and `get balance` and:

```
a !withdraw !Get_Account_Number(this_account) ?m: Money;
a !rtn_withdraw ...;
```

for the service `withdraw`.

`Get_Account_Number(this_account)` is an operation specified in the ADT where the sort `State_Account` is defined, returning the object identifier of an object instantiated from the class template `Account`. It corresponds to the individual variable $c_t.\iota$ reserved for the *'current object'*. Σ_S is the set of services whose names are `deposit`, `get balance` and `withdraw`. Specifically,

$$\Sigma_S = ran(\{\texttt{deposit}, \texttt{get\_balance}, \texttt{withdraw}\} \lhd named\ by)$$

Given these bindings of syntactic parts of the LOTOS expression to the formal parts of the class-template tuple and the identification of expressions of the form $c_t.\iota$, $c.y$, $c.y.service.name$ and $c_t.y.attribute.name$, the behaviour description $\mathcal{B}$ is given by the body of the process definition.

The interface function $\mathcal{I}$ is defined in LOTOS by identifying guarded expressions in the behaviour description. Consider, for example, the expression:

```
[not(zero_balance)] -> a !withdraw !Get_Account_Number ...
 ...
[zero_balance] -> exit(this_account)
```

This ensures that a client may only use the service `withdraw` if the balance is not zero. As ROOA deals with the analysis phase, a reference to the state of an object may be a symbolic reference. Therefore, in process `Account` we use the generalised LOTOS choice operator **choice** to cover the two possible situations expressed by the two guards `zero balance` and `not(zero_balance)`, whose values depend on the state of the object when the guard is evaluated at run time. The gate `a` is used to represent the channel of communication between `Account` and `Counter_Teller`.

The argument `this_account` defines the state of objects of class template `Account`, that is, it defines Σ_A and the collection of values at any time. In ROOA, this is defined by an ADT, as shown in Section 3.3.2. There is in each such ADT a distinguished operation always prefixed with the term `Make_` which has Σ_A as domain type and returns the sort of the object. It is used by the *object generator* to instantiate objects. Σ_A may be read directly off the domain expression in the LOTOS-type of the `Make` *whatever* operation. The ADT `Account_Type` shows the attributes in Σ_A as `Account_Number` and `Balance`. The operations defined in `Account_Type` are components of the methods used to perform the services offered (see Section 5.4.6).

Error Handling

The process `Account`, as it was specified above, brings us a problem: what happens when a client who has no balance in her or his account, asks for a withdrawal? There will be no process account to synchronise with it and therefore we would have a state of deadlock!

We could restrict the tellers to not accept withdrawals from clients whose accounts have no balance. However, this would not be a realistic solution and also it would impose that tellers would have to know that piece of information about each single account. In fact, what is needed here is to deal with abnormal errors, i.e. errors which may only occur if we take the pessimistic view of the problem.

In LOTOS these errors are not intellectually difficult to handle, as they are always specified in the same general manner. However, because our events do not deal only with variables, but also with values, we cannot propose a general event to catch all the erroneous situations, as proposed in [McC93], for example. Therefore, we have to deal with them in each single place they can appear.

In an early stage of ROOA, and for the banking system, we were dealing with these kinds of errors in this way. We soon realised that the benefits did not compensate the extra work. The specification was growing quickly and we were losing readability. It was really "slave" work with not enough compensations. For these reasons, and because we are in the analysis phase, we decided to leave this to be treated in the design phase.

Therefore, in ROOA, instead of defining the class template `Account` as shown above, we specify it as shown in Chapter 3.

5.4.5 Attributes

We define an attribute as a tuple

$$attribute = \langle name, value\_set \rangle$$

where $name \in dom(named\_by \triangleright attributes)$ and $value\_set \in set\_of\_values$ is the set of all values that the attribute may have.

In ROOA, an attribute is defined in a LOTOS ADT. In the banking example, we saw that the class template `Account` has two attributes: `Account Number` and `Balance`. These attributes appear as arguments of the operation `Make Account`.

5.4.6 Services

A service $s_i \in c_t.\Sigma_S$ is offered by all objects instantiating a given class template c_t. It is used by other objects to query or change the state of the object which offers it. Objects which are instances of the same class template may not directly use each others' services. (If an object A uses a service of object B, we say that objects A and B *communicate*. The communication channels between objects are determined statically, in the model, and correspond to gate `a` in the previous example.) Each service is defined by a unique name and by one or more methods. We define its type as follows:

$$service = signature \frown \langle method_1 ... method_n \rangle$$
$$signature = \langle name, obj\_id, in\_parameters, out\_parameters \rangle$$
$$in\_parameters \in attributes^{\,n} \times variable\_type\_pairs^{\,m} \times D$$

where $\frown$ is the concatenation operator on sequences, $D \in data\_types^{\,p}$ is a p-tuple of specific data types; and

$$out\_parameters \in Q$$

where $Q \in data\_types^{\,q}$; also $name \in dom(named\_by \rhd services)$.

We distinguish here the static structure of a method (simply its signature with its input and output parameters) from the description of how the method is used or how the value returned in the output parameters is calculated. The use of a method properly belongs to the dynamic description, as does the description of how the value is calculated, and must be described in $\mathcal{B}$. (TLA [Lam93] *actions* correspond to our methods.) Therefore, these features belong to the LOTOS description of the behaviour of the service. A *method* is defined by a LOTOS behaviour expression such as

$$method_i = \alpha;\ exit(ADT\_op(...))$$

where α is a collection of action denotations which can include invocation of services defined in other objects, for example. The *exit* construct indicates successful termination. This construct may have arguments which we use, in this case, to change the value of one or more attributes of the object, by invoking an operation defined in an ADT. This operation has the form:

$$ADT\_op : state\_sort, variable\_type\_pairs^{\,r} \rightarrow state\_sort$$

where *state_sort* represents the sort of the states of an object, e.g. `State Account` in the ADT above.

The behaviour of the method which specifies the service `withdraw` in the banking system, is given by the two following behaviour expressions:

```
( choice enough_money: Bool []
  [enough_money] ->
     a !rtn_withdraw !Get_Account_Number(this_account) !true;
     exit(Debit_Account(this_account, m))
[]
  [not (enough_money)] ->
     a !rtn_withdraw !Get_Account_Number(this_account) !false;
     exit(this_account)
)
```

where the *ADT_op* is `Debit_Account`. Each one of these two behaviour expressions define one method, each of which is used as an alternative of the other. Notice that there are methods which change the state of an object (called *modifiers*) and others which do not (called *selectors*). The selectors only return the value of one or more attributes, as happens with `Get_Balance`. In any case the *exit* construct always returns the state of the object, whether updated (as in `exit(Debit_Account(this_account, m))`) or not (as in `exit(this_account)`).

5.4.7 Object Generator

In ROOA, an object is instantiated from the class template by a special mechanism called an *object generator*. Although an object generator knows about the objects (identifiers) already created, it is mainly used to instantiate a class template.

$$object\_generator = \langle name, c\_t, create\_obj\_c\_t, object\_id\_set, \beta \rangle$$

An object generator has a name, $name \in dom(named\ by \triangleright object\ generators)$, and instantiates objects from the template by offering the service $create\ obj\ c\_t$ in a manner described by the behaviour β. It also contains a distinguished name $object\ id\_set$ for the set of identifiers of objects already instantiated from c_t.

When an object is instantiated by a call to $object\ generator.create\ obj\ c\_t$, it has an initial state. The initial state is defined in the behavioural description β. A behavioural description without a specified initial state is an invalid description. Most behavioural specification languages such as LOTOS, SDL and TLA either include or require such an initial state definition.

An object generator for class template `Account` looks as follows:

```
process Accounts[a](accs: Account_Number_Set) : noexit :=
  ( a !create ?acc_counter: Account_Number [(acc_counter notin accs)];
    ( Account[a](Make_Account(acc_counter, 0 of Balance))
    |||
      Accounts[a](Insert(acc_counter, accs))
    )
  )
```

where `Accounts` is the name of the object generator and `accs` is the set of identifiers of objects already instantiated from class template `Account`. The service whose signature is:

```
a !create ?acc_counter: Account_Number [(acc_counter notin accs)];
```

corresponds to $create\_obj\_c\_t$ and it uses value generation to generate an object identifier of sort `Account_Number` which is then used to instantiate the class template by executing

```
Account[a](Make_Account(acc_counter, 0 of Balance))
```

The operation `Make_Account` is defined in the ADT `Account_Type` and it creates a value of the sort `State_Account`. The initial state of the object account created is the result given by `Make_Account`.

The behaviour β is given by the body of the process definition. The object generator is defined recursively so that we can continue to create objects.

5.5 Defining the Concepts of Object-Oriented Systems

Certain concepts of object-oriented methods are not properly part of the object model, as described for example in [RBP +91], but are part of the system design that derives from a given object model. Most attempts to define the semantics of object-oriented systems concentrate on objects, classes, inheritance and aggregation, and are properly part of the system design, not the model. Our approach produces a model which fits closely with the object model and, at the same time, uses the power of LOTOS to describe behaviour.

5.5.1 Object

An object in the design represents what 'runs' in the implemented system. A class template is a definition which is used to create objects. An object is an instantiation of a class template. The class template includes the description of the generic behaviour of an object, as well as defining all attributes which the object may have. In order to know all about an object it is sufficient to know about only which template c_t it was instantiated

from, and its identifier (the value of the attribute $c\ _t.obj\_id$). The object's behaviour must be derived from $c_t.\mathcal{B}$. Hence we define

$$object = \langle ident, class \perp template.name \rangle$$

where $ident \in dom(named \perp by \rhd objects)$.

5.5.2 Behavioural Constraints vs. Behavioural History

In the definition of *object* in Section 5.5.1, the object itself is rather bare. Objects have a history of behaviour, which may be different for each object, and furthermore is not derivable from the class template. Where, one may ask, is this behavioural history in the definition of object?

The answer is that the history of behaviour of an object, no matter how important for the operation of the object in the system, is not properly part of a *semantic* account of class template, object, and their relation. In contrast, an account of how the *constraints* on an object's possible behaviour are obtained *is* properly part of the semantics. $c\ _t.\mathcal{B}$ is a constraint expression, with lots of free variables. We describe in Section 5.5.3 how the behavioural constraints on an object are obtained from the constraint expression in the template.

This position may be compared with that of semantics of programs. Compare objects with programs. A specification includes the constraints on a program's behaviour. An actual behaviour of the program, or a partial behaviour (a *history*) is described by a (partial) trace. The specification describes the set of all possible traces. Each individual partial trace that satisfies the specification, and which may or may not occur during the lifetime of the system, is not considered part of semantics. It is up to the program whether it wants to keep around such information about its previous behaviour. In many object-oriented systems, this may be important for auditing, just as in previous designs in the before object-orientation era, keeping history variables around for the same purposes was also considered important. But it is not part of semantics. It is part of programming.

5.5.3 Deriving the Behaviour of an Object from the Class Template

Let `ob` be the *'current object'* and let its identifier `ob`.*ident* be *oid* for notational simplicity. The behaviour of `ob` can be described by $\mathcal{B}[c\ _t.\iota/oid]$ where $\mathcal{B}[c_t.\iota/oid]$ is the expression obtained by substituting *oid* for every occurrence of the distinguished individual variable $c_t.\iota$ in $\mathcal{B}$ [Lad94]. $\mathcal{B}[c_t.\iota/oid]$ still contains individual variables, say for example simply $c.y$, for other objects whose services `ob` may use. Since the expression still has an unbound individual variable $c.y$, it does not yet describe concrete behaviour. The description is supposed to say what behaviour is allowed between `ob` and other objects instantiated from c.

Suppose that $\texttt{ob}_1$ and $\texttt{ob}_2$ are precisely the instantiated objects from template c, with identifiers $\texttt{ob}_1.ident$ and $\texttt{ob}_2.ident$ respectively, which we denote $c.oid_1$ and $c.oid_2$ for simplicity. The required description of `ob`'s interaction with one specific other object is obtained when any value of another object identifier instantiating a template c is substituted for the variable $c.y$ appearing in $\mathcal{B}[c\ _t.\iota/oid]$. The object `ob` may engage in its c-object interactions with $\texttt{ob}_1$, and also with $\texttt{ob}_2$. Thus the two expressions $\mathcal{B}[c\ _t.\iota/oid][c.y/c.oid_1]$ and $\mathcal{B}[c_t.\iota/oid][c.y/c.oid_2]$ describe the interactions in which `ob` can engage.

A combination of these behaviour expressions expresses the behaviour of ob in terms of all the other c-objects ob $_1$ and ob$_2$. The combination of expression $\mathcal{B}[c\ _t.\iota/oid][c.y/c.oid\ _1]$ with expression $\mathcal{B}[c\ _t.\iota/oid][c.y/c.oid\ _2]$ may take different forms according to the specific language in which $\mathcal{B}$ is written. In LOTOS, the combinator is '[]' and the combined expression is:

$$\mathcal{B}[c_t.\iota/oid][c.y/c.oid\ _1]\ []\ \mathcal{B}[c_t.\iota/oid][c.y/c.oid\ _2]$$

If a logical language is used, then the combination is *logical disjunction*, yielding

$$\mathcal{B}[c_t.\iota/oid][c.y/c.oid\ _1] \vee \mathcal{B}[c_t.\iota/oid][c.y/c.oid\ _2]$$

This states that ob can engage in $\mathcal{B}$ behaviour with *either* ob $_1$ *or* ob$_2$.

Although this operation has been expressed with one template and two instantiated objects for simplicity, it generalises directly to multiple class templates and instantiated objects. Whatever its specific form, this syntactic operation of *substitution + combination* yields the behavioural description of object ob at any state of the system. As other objects are created, a similar operation must be performed: say ob $_3$ is newly instantiated with identifier ob$_3$.*ident*, denoted $c.oid\ _3$ for simplicity. The expression $\mathcal{B}[c\ _t.\iota/oid][c.y/c.oid\ _3]$ must be formed and *combined* with the other behaviour expressions to yield the behavioural description of object ob in the new environment.

Note that the expressions described here need not actually be formed. It is sufficient that a procedure exists to obtain a precise description of the behaviour of an object in its run-time environment from the behaviour expression $c\ _t.\mathcal{B}$ in the class template $c\ _t$. We have given such a procedure.

5.5.4 Deriving the Behaviour of an Object in LOTOS

To illustrate the procedure in LOTOS, we introduce part of a process definition which specifies the class template Counter Teller:

```
process Counter_Teller[t, a](id: Id_Tellers) : noexit :=
  ( t !get_balance !id ?acc_nr: Account_number;
    a !get_balance !acc_nr ?b: Balance;
    t !rtn_balance !id !acc_nr !b;
    exit(id)
   []
     ...
    ) >> accept id: Id_Tellers in Counter_Teller[t, a](id)
  endproc
```

Objects of the class template Counter Teller invoke services of objects of the class template Account. Let Counter Teller be the class template $c\ _t$, the value id be the object identifier of the current object, while the LOTOS-variable acc_nr: Account_Number corresponds to the individual variable $c.y$.

An expression describing the behaviour of an object instantiated from Counter Teller, having, say, identifier 125, i.e. the expression corresponding to $\mathcal{B}[Counter\ \ Teller.\iota/125]$, is:

```
t !get_balance !125 ?acc_nr: Account_number; ...
```

Counter teller number `125` can communicate with any object instantiated from the class template `Account`. Suppose we create precisely two accounts with identifiers 35467 and 35468. The complete behaviour of counter teller `125` when communicating with any of the two accounts, asking for any service, is given by the combination of the services offered by `Counter_Teller`. We substitute 35467 and 35468 for the expression `acc nr`, and combine with the choice operator [] to obtain

$$\mathcal{B}[Counter\_Teller.\iota/\texttt{125}][acc\ nr/35467]\ []\ \mathcal{B}[Counter\_Teller.\iota/\texttt{125}][acc\ nr/35468]$$

Notice that the behaviour of counter teller `125` changes with time, according to the services executed. Suppose we created the two accounts, one subsequently to the other. When communicating with account number 35467, before account 34568 was created, counter teller `125`'s behaviour is described by: $\mathcal{B}[Counter\_Teller.\iota/\texttt{125}][acc\ nr/35467]$. After account 35468 is created, the behaviour expression changes to include in the combination the expression $\mathcal{B}[Counter\_Teller.\iota/\texttt{125}][acc\ nr/35468]$.

This informal description has illustrated how a description of the behaviour of an object in a specific system may be obtained by purely syntactic means from the behaviour description $\mathcal{B}$ in the class template. The description has been given in a generic way and the particular syntactic operations needed to effect this for the given behavioural specification language LOTOS have been used as illustration. This shows how the behaviour of object `ob` instantiated from template c_t is determined by the behavioural description $c_t.\mathcal{B}$. A formal definition of `ob` needs to specify no extra behavioural component beyond that contained in $c_t.\mathcal{B}$.

5.5.5 *Is_instance*, Classes and Other Concepts

The *is_instance* relation is a binary relation between objects and the class templates of which they are instances. When an object is created, according to the semantics it is a pair of names $\texttt{ob} = \langle identifier, class\_template.name \rangle$. The image of `ob` under the *named by* relation [Spi89, page 123] ($named\_by \circ \texttt{ob}$), is thus the pair consisting of the object along with the class template it was instantiated from. This pair is an element of *is instance*. If one requires this relation for any purpose, it is necessary to require of the object-generator that when it creates an object `ob`, it also adds ($named\_by \circ \texttt{ob}$) to *is instance* (which is defined to have value $\emptyset$ when no objects have yet been created). One may then straightforwardly define the class generated by c_t, $class(c_t) = dom(is\_instance \rhd \{c_t\})$.

An account of the concepts of an object-oriented system will also include a semantics for *message-passing* and for *inheritance*, which require some more detailed attention than the rather simple definition of class and *is instance* above.

5.6 Conclusions

In this chapter, we have given a simple denotational semantics for the concepts of object-oriented analysis, which include *class template*, *service*, *attribute*, *behaviour*, *object generator* and *visibility*. We described the form required of a *behaviour* expression in a class template, so that the behaviour of any object instantiated from that template is precisely specified. We employed the notions of *individual variable* and *substitution* from the syntax of logic to explain in general terms how this behavioural description was transformed. We

illustrated this transformation with ROOA templates and behaviour expressions written in LOTOS.

Chapter 6

The Rigorous Object-Oriented Analysis Method

6.1 Introduction

The Rigorous Object-Oriented Analysis (ROOA) method takes the static properties of a system captured in an object model and the dynamic and functional properties described in the original set of requirements and produces a formal object-oriented analysis model. The object model can be built by any of the existing OOA methods.

The formal object-oriented analysis model:

1. Formalises the object model describing each class template and relationship in a mathematical manner.
2. Adds state information and behaviour to each class template and describes the order in which the events occur.
3. Shows the message connections between objects and the information passed during communication.
4. Defines the behaviour of the whole system by putting together its classes.
5. Is formal and executable, and therefore rapid prototyping can be used to check the conformance of the specification against the original requirements and to detect inconsistencies, omissions and ambiguities in the original requirements.

The formalisation of the information in an object model can be done semi-automatically, i.e. it requires some decisions, but most of the work is fairly straightforward and is similar for each class template. However, identifying the behaviour of each class template, the events and their order and the information passed during message communication is not trivial. Existing OOA methods propose two extra models (see Figure 6.1) to capture the behaviour of a system (the dynamic model) and the transformations of the data (the functional model). They then end up with three models, each of which shows different aspects of the system and which are difficult to integrate and keep consistent. Furthermore, the dynamic model is difficult to understand since it does not give an integrated view of the behaviour of the system. It is usually composed of a set of state transition diagrams, one for each class template.

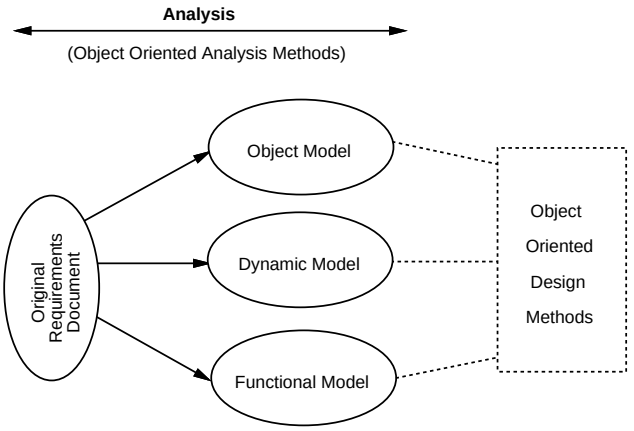

Figure 6.1: The models built by many object-oriented analysis methods

In contrast, the ROOA method produces a single formal model which integrates the static, dynamic and functional properties of a system. The ROOA model is primarily a dynamic model, but it preserves the structure of the object model.

6.2 The ROOA Method

ROOA gives us an integrated view of the system, showing at the same time its static, dynamic and functional properties. The formal description technique (FDT) we have chosen is LOTOS. As LOTOS has a precise syntax and mathematical semantics, the resulting model is formal and unambiguous. Moreover, as LOTOS is executable, the model is executable, and so prototyping can be used to give immediate feedback to the clients who can check if the prototype exhibits the intended behaviour.

ROOA uses a stepwise refinement approach for the development and for validation of the specification against the requirements. The development process is iterative. Different objects may be represented at different levels of abstraction and the model refined incrementally.

ROOA involves three tasks:

1. Build an object model.
   The construction of the object model is performed by applying any of the existing object-oriented analysis methods, such as OOA by Coad and Yourdon, OOSE by Jacobson, OMT by Rumbaugh *et al.* [CY91a, Jac92, RBP $^{+}$91].

2. Refine the object model.
   To refine the object model we start by guaranteeing that it includes interface objects, attributes, and static relationships; then we identify initial dynamic behaviour by identifying services and message connections; finally, we structure the object model by identifying subsystems.

3. Build the formal LOTOS OOA model.
   The LOTOS formal model specifies the object model, gives the dynamic behaviour

of each object and of the whole system, shows the message communications between objects in the system and also models the information passed when objects communicate.

ROOA acts as the central part of the analysis phase, but it interacts with requirements capture and can provide the starting point of the design phase. Figure 6.2 illustrates ROOA in the context of the software development life cycle and shows how the various tasks are connected.

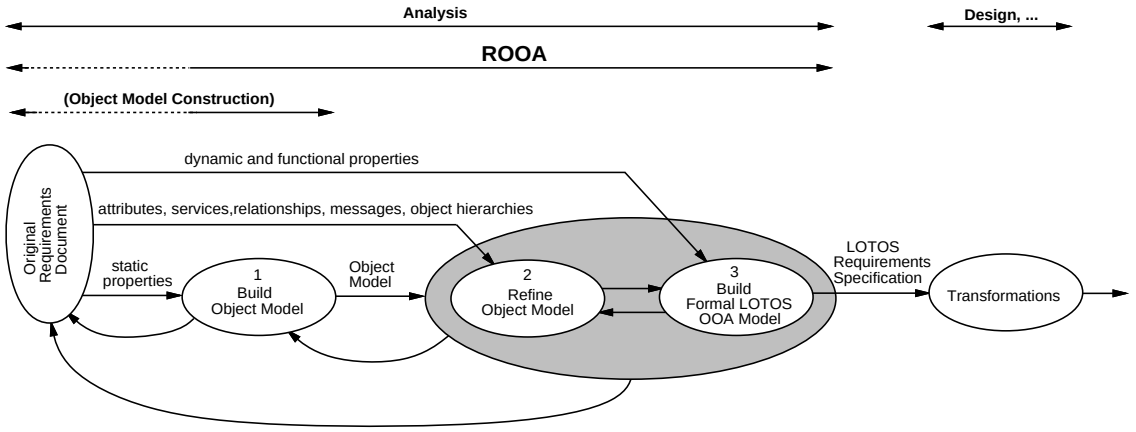

Figure 6.2: Context of ROOA in the development life cycle

The object model construction is, of necessity, informal. It is performed by reading through the requirements document, interviewing the clients (or the users), etc., finding the objects of the system and the relationships between them. We use the part of an OOA method that builds the object model to perform it. The application of the next two tasks of ROOA may lead us to change the object model, and if we find any omissions or inconsistencies we also change the requirements document where the problem we are analysing is stated. We can use ROOA with prototyping to analyse the system incrementally.

The resulting ROOA specification can then be used as the first stage in a software development trajectory where the requirements specification is transformed into a design specification either by using correctness preserving transformations or by using prototyping to ensure that the two specifications conform to one another.

Figure 6.3 gives a view of the two main tasks of ROOA, *Refine Object Model* and *Build Formal LOTOS OOA Model*, showing their composition.

The main goal of ROOA is to produce a formal LOTOS object-oriented analysis model. To accomplish this, ROOA has to refine the object model produced by one of the OOA methods and formalise it. It then has to identify the dynamic properties of each class template and of the overall system in the original requirements and formally specify them. Finally, it has to identify the information passed during communication, also in the original requirements, and formally specify it. However, if the starting point is the result of a separate team's application of one of the OOA methods, dynamic and functional models may already have been produced. In this situation, ROOA would integrate the information spread among the three models and give a formal and executable specification of the system. Given the existing dynamic and functional models, the application of the ROOA tasks would be much faster.

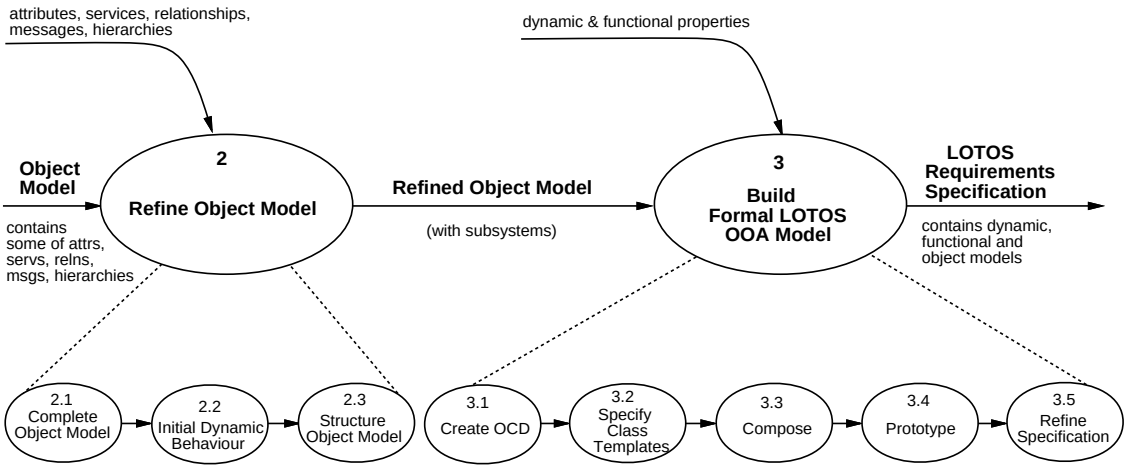

Figure 6.3: Core of ROOA

Analysts know how important it is to involve the clients (or the users) of the system in the development process. Some can argue that FDTs should not be used so early in the software development life cycle, since they are not easily understood by most clients. That may be true, but there are other advantages of using FDTs as early as possible to develop software, especially if the resulting specification is executable. Apart from the advantages already mentioned (such as having a formal description of the system and using prototyping to find inconsistencies, omissions and ambiguities early) there is also the advantage of increasing the analyst's confidence in the system. This is due to two main reasons:

- By using FDTs, analysts have to understand every "corner" of a problem in order to specify it. The use of informal techniques allows them to be vague in the description and so may unintentionally avoid certain characteristics of the system that they do not fully understand. By the time they finish the analysis using FDTs they will know much more about the system than they would know if they were using informal techniques. This knowledge helps them to present their idea of the system to the clients and understand their explanations.

- If the analysts use an FDT which produces executable specifications they can also use rapid prototyping for checking the specification against the requirements. The possibility of using prototyping gives them confidence in the specification they are developing. By using ROOA, they also have the possibility of developing the system incrementally, using components which have already been verified.

These advantages compensate for the difficulty the clients may have in reading the specification. The simulation tools can be used to show the system to the clients. For our method we suggest the analysts use the simulator SMILE [EW93] for their work since, as we said in Chapter 2, it supports non-determinism and value generation. However, this tool is perhaps too complicated to be used to show the system to the client. The simulator HIPPO [vEVD89] does not offer as many possibilities to check the specification, but it has a much simpler interface and so we advise it to be used when the specification is to be shown to the client.

The final purpose of this chapter is to describe each task of the ROOA method. This description is accomplished by using, as a case study, a running example. The example we have chosen is the automated banking system whose requirements were presented in Chapter 3. ROOA was first applied to this problem with Coad and Yourdon's method (Figure 3.2 depicts the object model obtained). Later, we have applied ROOA with OMT to the same problem. The goal of this exercise was to test how ROOA would behave in conjunction with a different method. This turned out to be an interesting exercise, as it showed us important differences between the two OOA methods with regard to the information their resulting object models contain. The identification of those differences led us to improving ROOA to better integrate object models produced by most OOA methods. In the following section, as we present each task, we give the results of the application of ROOA together with OMT.

6.3 The ROOA Process

An object model shows class templates and concentrates on the static relationships between objects. The integrated analysis model produced by ROOA specifies a system's dynamic behaviour and the data transformations as well as the static relationships. It typically involves many instances of each class template. To avoid being overwhelmed with detail in the first iteration of ROOA, only a single instance of each class template is considered, i.e. we focus on the concept of a single typical object of a class. For this reason, the names given to the documents and diagrams use the term object (e.g. object model, object communication table and object communication diagram) and the discussion in the following sections is in terms of objects rather than classes.

In later iterations, the model is generalized so that we deal with classes rather than objects. However, the term object is kept for the diagrams and documents, even though, by the end of the last iteration, they reflect the more general concept of class template and class.

Task 1: Build an Object Model

An object model shows the class templates that compose the system and the relationships between their objects. The objects can be found by looking for physical entities and concepts in the problem domain. Not all the objects are explicit in the system requirements document; some are implicit to the problem domain or the general knowledge of the real world. In general it is not difficult to identify objects, but it is difficult to select which of them are relevant to the system. Our goal is to identify the objects which are essential throughout the system's development life cycle.

The construction of the object model can be considered as a separate task from the other ROOA tasks, and it can be accomplished by a different team. During application of our method, the object model may be modified. The advantage of starting with an object model produced by any object-oriented analysis method is that we can build on the work which has already been done to identify objects.

The object model is the main focus of most OOA methods. This brings no problems, if the problem we are modelling is primarily static. However, there are other problems which have a preponderant dynamic component. For this kind of problem, we may need

to use techniques such as Event Trace Diagrams (ETDs) [RBP +91] to help us build the object model.

An example of an object model for the problem described in Section 3.3 and which was built by using OMT [RBP +91] is depicted in Figure 6.4.

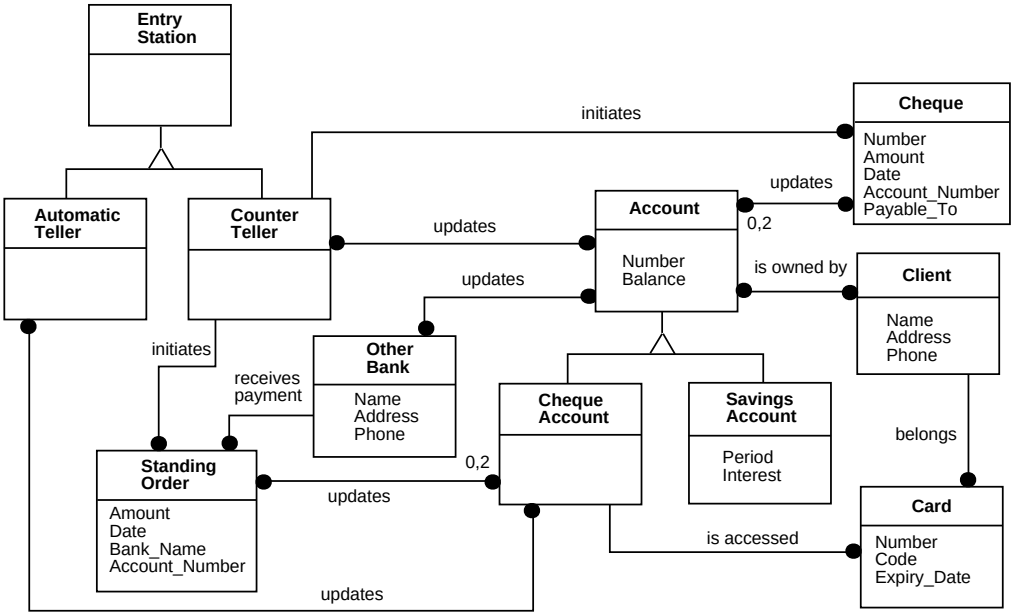

Figure 6.4: Object model produced by the OMT method

Task 2: Refine the Object Model

Many OOA methods are mainly concerned with the identification of *entity objects*. Entity objects correspond to those objects that support the information that the system must keep and maintain. According to Jacobson, there are however other kinds of objects: *interface objects* and *control objects* [Jac92]. Interface objects are the objects that the *actors* use to communicate with the system, ask it for services and receive answers from it. (Actors are the external objects, clients or users, that interact with the system.) They transfer the actors' actions into system events and the system's answers into something that the actor can read and understand. The control objects deal with the functionality that does not fit into any of the entity or interface objects. The entity objects will exist in the initial object model, but the interface objects may not.

During this task, we add interface objects to the object model. Next we refine the resulting object model so that each class template is described with attributes and services. Then we define static relationships and message connections between objects, and finally, we start structuring the system. To structure the system, we use aggregates and subsystems. Aggregates are complex objects, built from simpler objects, which may already have been identified in Task 1. Subsystems are also groupings of objects, but they do not enjoy the status of being an object. These groupings are often found by minimizing the static relationships between objects. We wish to identify subsystems by taking into consideration message connections as well as static relationships.

Therefore, before we start producing the LOTOS formal model, we want our object model to reflect the static properties and the dynamic dependency between objects. This is achieved by guaranteeing that it incorporates:

1. Interface objects.
2. Static relationships.
3. Attributes and services in each class template.
4. Message connections.

Task 2.1: Complete the Object Model

During this task we start adding interface objects to the object model and then we add more information to each class template. In object models created using some OOA methods [RBP+91], the class templates are defined only by name and a list of attributes. In others [CY91a] we also have services and message connections. We propose that a class template can only be fully understood if it is defined by a name, a list of attributes and a list of offered services. Moreover, in our object model the class templates are related by static relationships (and message connections which are added in Task 2.2.2).

Task 2.1.1: Add Interface Objects

Interface objects model behaviour and information that is part of the system interface with the system's environment. Thus, everything in the system that is concerned with an interface is placed in interface objects. An example of an interface object is `Automatic Teller`.

Task 2.1.2: Add Static Relationships

A static relationship between two (or more) objects means that one object "knows about" the other (or, if the relationship is bidirectional, they "know about" each other). There are three kinds of static relationships:

1. Conceptual relationships.
2. Aggregation.
3. Inheritance.

Relationships are present in most object models given by OOA methods. However, the method proposed by Berard [Ber89, Ber93] puts the objects together in the object model by showing the visibility between objects (which correspond to message connections), without defining relationships. Also, some relationships in OMT turn out to be message connections.

Task: If static relationships are not already in the object model, identify them and add them to it.

Task 2.1.3: Add Attributes (and Services)

Attributes can be viewed as the components (elementary or not) which make up the state of an object, although it is possible for an object not to have attributes. The services offered by an object constitute the mechanism that allows other objects to change or query that object's state information. Some authors argue that as services will change during the design phase, it is not relevant to add them to the objects in the analysis phase [Jac92, RBP +91]. We believe that most of the services identified during this phase will be retained with few changes during the design. Hence, we propose that for an object to be fully understood, its definition should include its attributes, if any, and its services.

To identify the services offered by each object, we can follow two steps. First, place ourselves inside each object and, according to the original requirements of the system, identify the services each of the objects has to offer to its environment so that the object's attributes can be interrogated and updated. Further services may be identified when we deal with message connections.

Task: If attributes and services are not already in the object model, identify them and add them to it. The services may be left until Task 2.2.

Task 2.2: Initial Identification of Dynamic Behaviour

An object model describes the static properties of a system. To capture and record dynamic behaviour from the information given in the informal requirements, we proceed according to the following steps:

1. we model the behaviour that the environment requires from the system as a set of *interface scenarios*, related to the *use cases* of Jacobson [Jac92]; then,

2. we build ETDs to show dynamic behaviour of the system (they may already exist from Task 1); and finally,

3. we collect the information in the ETDs in an Object Communication Table.

Notice that interface objects belong to the system, but interface scenarios do not.

Task 2.2.1: Define Interface Scenarios

Interface scenarios model the interaction of a system with its environment, i.e. they show a series of services (requests and responses) that the actors (clients or users) can require from the system. Each one, modelling different parts of the functionality of the system, can be seen as a list of calls to the services offered by the interface objects together with the expected responses.

By using interface scenarios together with the object model, we can understand the system's functionality and dynamic behaviour.

The interface scenarios will be modelled in LOTOS in Task 3. They play an important role during the prototyping of the formal specification.

Task 2.2.2: Define ETDs and Start Object Communication Table

Starting with an interface scenario event, we trace through the object model identifying the object interactions required to satisfy the requested behaviour and record the information in an ETD. The events in the ETDs may suggest additional services to add to the class templates in the object model. Some services could have already been identified in Task 2.1.3.

Consider once more the banking example. In Chapter 3 we showed how to map OOA constructs into LOTOS and illustrated some of the LOTOS code for the banking system. Here, we will consider one of the class templates which we found, as we will show during this chapter, more difficult to model: `Cheque`. In Figure 6.5 we present a simple ETD showing the events exchanged between objects when a `deposit cheque` service is required.

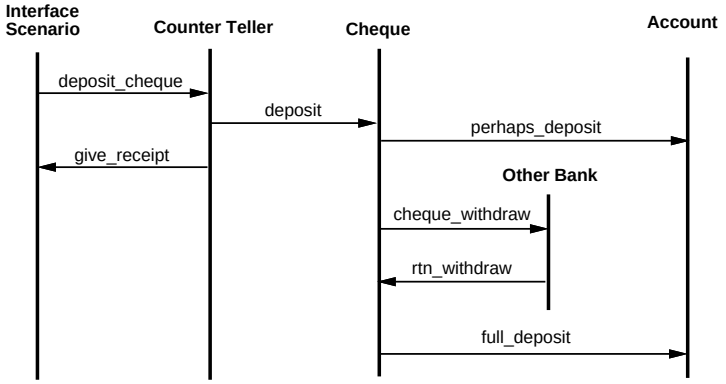

Figure 6.5: ETD for depositing a cheque belonging to another bank

Other ETDs showed us that `Cheque` should offer two services: `deposit` and `withdraw`. By drawing ETDs we could understand better what a 'cheque' was. ETDs help us reason about the behaviour of objects without having to worry yet about formality. While drawing this ETD, we discovered that we needed to include two extra services in `Cheque Account`: `perhaps_deposit` and `full_deposit`. Together with these services, we identified an extra attribute in `Account`, which is `Balance Pending`. When depositing a cheque, `Balance Pending` takes the amount to be deposited, by using the service `perhaps deposit` and, when the cheque is cleared, `full deposit` adds the value of `Balance Pending` to `Balance`. In Chapter 3, we did not identify these services while building the object model by using Coad and Yourdon's method (see Figure 3.2).

The information from the different ETDs (and from the object model) is then collected in an Object Communication Table (OCT). Eventually, this table will be composed of five columns, but now we are only building the first two columns. In the first column we list the class templates that form the object model and in the second column we list the services offered by each class template. If the services are given in the object model, column one and column two can be filled in directly (see Table 6.1).

Task: Define interface scenarios and ETDs, add services, if appropriate, to the class templates in the object model. Fill columns one and two of the OCT.

Task 2.2.3: Add Message Connections

A message connection shows a processing (dynamic) dependency between a client and its server. It is defined as single arrow, not a double arrow. Therefore, if A is a client of B and B is a client of A, we draw two arrows, as shown in Figure 6.6.

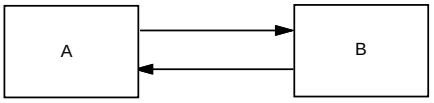

Figure 6.6: Message connections between A and B

During this task we fill the third and fourth columns of the OCT. In the third column (*Required Services*) we list, for each service offered by a class template in column one, the services that that class template requires from other class templates to accomplish that particular service. The notation <*class template.service*> is used to refer to the required *service* defined in *class template*. In the fourth column (*Clients*) we list, for each service offered in column one, the class templates (clients) which require that service. For each offered service we may have a list of clients. Note that the clients of the interface objects are the interface scenarios.

To help us to identify the required services we use the interface scenarios and the ETDs. For example, starting with an interface scenario event, we can follow complete paths of functionality in the system, filling the table and creating corresponding message connections as we trace instances of the class templates in the object model. As we trace through the object model "simulating" threads of its functionality, new services may be identified which should be added to the appropriate class template and inserted in the second column of the table. Notice that by performing this step we are also checking the necessity for each service in the object model.

ETDs are a major help in showing message connections, as they show the events exchanged between instances of class templates.

For the object model in Figure 6.4, and considering each class template with its services, the first four columns of the OCT would be filled as shown in Table 6.1 and the information added to the object model, as shown in Figure 6.7.

When we construct this table we may identify information that we are not yet able to describe. In general, if a service requires more than one service from other class templates, there is an order in which the required services occur. If this order is sequential, it can be given by the order in which we fill column three. There are, however, situations where the services required are alternatives. For example, the class template `Cheque` offers `deposit`. If the cheque is drawn on our bank, then we can withdraw the amount from the account where it is drawn and credit the payee's account. However, if the cheque is drawn on another bank, before we credit it to our client's account, we must query `Other Bank` to know whether or not funds are available to cover the cheque. Part of this information may be given in the ETDs which can be used later as a guideline to build the LOTOS specification.

The first two columns in the OCT duplicate information given in the object model. Tools such as ObjecTool permit extra information to be recorded about attributes, services, etc. Such information is not presented diagrammatically in the object model, but is available in textual form. Instead of building the OCT by hand, we can record, for

Class Templates	Offered Services	Required Services	Clients
Entry_Station (ES)	withdraw_cash	Account.withdraw	Interface Scenario
Counter_Teller (CT)	open_account	Account.create	Interface Scenario
	close_account	Account.remove	Interface Scenario
	deposit_cash	Account.deposit	Interface Scenario
	give_balance	Account.get_balance	Interface Scenario
	deposit_cheque	Cheque.deposit	Interface Scenario
	ask_transfer	Account.withdraw Account.deposit OB.send_transfer	Interface Scenario
	set_standing_order	SO.create	Interface Scenario
	cancel_standing_order	SO.cancel	Interface Scenario
Automatic_Teller (AT)	mini_statement	Cheque_Account.print_mini_stat	Interface Scenario
Other_Bank (OB)	receive_transfer	Account.deposit	Interface Scenario
	send_transfer		CT, SO
	cheque_withdraw		Cheque
	remote_withdraw	Account.withdraw	Interface Scenario
Standing_Order (SO)	create		CT
	cancel		CT
	debit	Account.withdraw Account.deposit OB.send_transfer	internal
Cheque	withdraw	Account.withdraw	CT
	deposit	Account.withdraw Account.deposit Account.perhaps_deposit OB.cheque_withdraw Account.full_deposit	CT
Account (A)	create		CT
	remove		CT
	deposit		CT, Cheque, OB, SO
	withdraw		CT, ES, Cheque, OB, SO
	balance		CT
Cheque_Account (CA)	print_mini_stat		AT
	perhaps_deposit		Cheque
	full_deposit		Cheque
Savings_Account (SA)	credit_interest		internal
	update_date		internal

Table 6.1: OCT with class templates, services offered, services required and clients

each service, its clients and the lower level services it requires. A translator has been constructed by Clark [Cla94a] which takes the textual output from ObjecTool and creates the corresponding OCT.

The message connections are drawn in the object model according to the OCT. If one class template requires services defined in more than one class template (server), then there is an arrow starting from the first class template reaching each of the servers. Only one message connection is drawn from one particular class template to another class template, independently of the number of services the first class template needs from the second.

Task: Complete columns three and four of the OCT and add, if necessary, message connections to the object model.

Task 2.3: Structure the Object Model

Grouping class templates into subsystems or into aggregates is necessary when we are dealing with large complex systems.

This task is difficult to accomplish and so we cannot expect to do it completely and correctly in the first iteration. The low level class templates in the object model often remain almost unchanged during the development, but the high level structure is less stable. Our suggestion is to do only what is obvious to begin with, and then come back to it as our knowledge about each individual class template increases. We use aggregates and subsystems to structure a large system. The fundamental difference between an aggregate and a subsystem is that, while the components are an intrinsic part of the aggregate, and the aggregate is itself an object from the problem domain, a subsystem is merely a grouping of related class templates.

During the first iteration, only *obvious groupings* are identified. Suitable candidates are:

- Class templates that participate in an aggregation relationship; they form an aggregate.
- Class templates that participate in an inheritance relationship; they form a subsystem.

Grouping other class templates should be left until later iterations, when we have a better knowledge about the system. However, when we identify subsystems we have to make sure that their components are logically related, highly coupled and that the interaction between subsystems should be low. Other candidates to form subsystems, which may only be grouped in later stages, are:

- a set of clients which use the same servers;
- a set of servers which have the same clients.

We show groupings in the object model by surrounding the class templates by a rectangle with dotted lines. The OCT should be changed to encode this information. The changes include using the name of the subsystem or aggregate in the first column of the table, instead of the component's name, and using the name of the subsystem or aggregate followed by the name of the component between brackets in any other place where that component is mentioned. For example, instead of considering `Counter Teller`, `Automatic_Teller` and `Entry Station` independently, we should only deal with with `Teller`. (The same can be said about `Account`, `Savings Account` and `Cheque Account`.) These changes are shown in columns one to four in Table 6.2.

Figure 6.7 represents the refined object model first shown in Figure 6.4.

Task 3: Build the LOTOS Formal Model

The advantage of building a LOTOS formal model is that we end up with a single model which:

- Models the static, dynamic and functional aspects of the system.
- Has a formal semantics.
- Is executable.

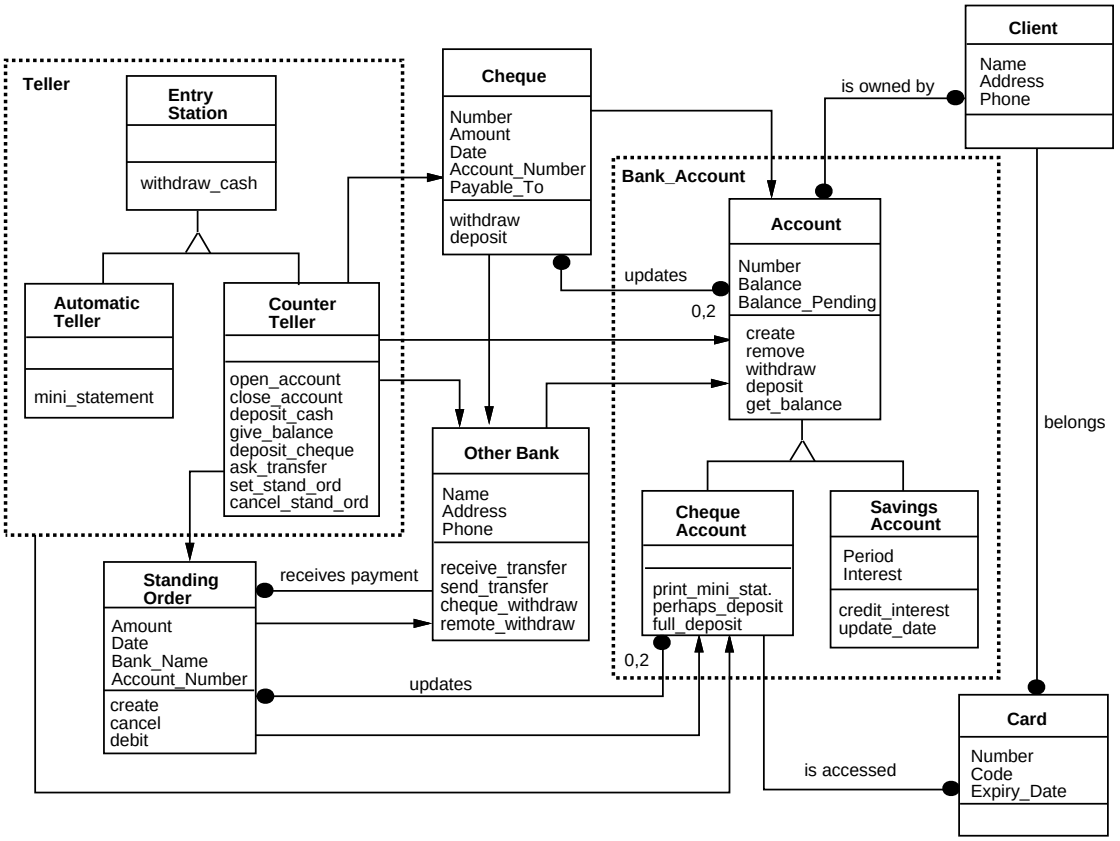

Figure 6.7: Refined object model

The resulting model acts as a formal requirements specification. Since a LOTOS specification is executable, rapid prototyping can be used to discover and correct inconsistencies, omissions, contradictions and ambiguities while we are still doing analysis. Moreover, the resulting model incorporates the characteristics of object-oriented systems, since it considers a system as a set of concurrent objects where message passing is modelled by objects synchronizing on an event during which information may be passed.

We start by building an object communication diagram, which shows the system as a set of communicating objects. It gives the structure of the LOTOS specification. Then, we model each class template in the object model in LOTOS and we add to it its behaviour. Next, by following the structure of the object communication diagram, we compose the objects (instances of the class templates) by using parallel operators and we prototype the specification. Finally, we refine the specification.

Task 3.1: Create an Object Communication Diagram (OCD)

The OCD is a graph where, in the first iteration, each node represents an object and each arc connecting two objects represents a gate of communication between them. In later iterations, the diagram is generalised to deal with multiple instances of the same class template. In the beginning, some of the objects may not be connected by arcs to the rest of the diagram. As the method is applied, these objects will either disappear or be

connected to the others, and new groupings may appear, refining the diagram.

The central feature of the final LOTOS specification is that it is primarily a dynamic model, although it contains all the static information given in an object model. A major task in the creation of the formal LOTOS model is the transformation from a static to a dynamic model. The OCD and the OCT are intermediate structures created to help in the transformation. The OCD shows the structure of the object model which then is preserved in the structure of the behaviour expression in the LOTOS dynamic model.

During this task we also complete the OCT that we started building in Task 2.2.2, by adding to it the column *Gates*. This column will give the name of the gates that the objects in column one and column four use to communicate between each other. The fifth column of the OCT is then used to label the arcs in the OCD.

The following algorithm shows how to construct the OCD from the OCT and the object model:

1. Draw the nodes.
   In the first iteration, take an instance of each class template in column one of the OCT and for each one draw a node in the OCD. For each object grouping already identified, proceed in a similar way, considering the composite object or subsystem as a node and showing each object component as an inner node of the first. Again, each inner node corresponds to an instance of each component class template.

2. Connect nodes by arcs.
   For each message connection between two class templates in the object model, draw an arc between the two corresponding nodes in the OCD. If a message connection to a component of a composite object or subsystem is defined, draw the arc in the OCD to the higher level node. At the end of the first iteration, some nodes may be unconnected to the rest of the diagram.

3. Label arcs.
   Repeat this step for each class template in the first column of the OCT.

   (a) Complete column five in the table.
   Looking at the second (*Offered Services*) and fourth (*Clients*) columns:
   - Give the same gate name for the object communications which require the same set, or subset, of services; i.e. where there is an overlap between the set of services required by different clients.
   - Give different gate names for object communications which require a different set of services, i.e. where there is no overlap between the set of services required by each client.
     These two rules are the result of requiring that an object cannot use the same gate to communicate with both an object at the same level of abstraction and another object at a different level of abstraction.
     Table 6.2 shows the gate names for the banking example.

   (b) Give gate names to the arcs in the OCD as follows:
   For each class template in column one of the OCT, identify the arc in the OCD that connects the corresponding object to each of the clients in column four. The name of the arc is the name of the gate given in column five to the corresponding server and client pair.

Class Templates	Offered Services	Required Services	Clients	Gates
Teller	open_account(CT)	BA.create	Interface Scenario t	
[ES + CT + AT]	close_account(CT)	BA.remove	Interface Scenario t	
	deposit_cash(CT)	BA.deposit	Interface Scenario t	
	withdraw_cash(ES)	BA.withdraw	Interface Scenario t	
	give_balance(CT)	BA.get_balance	Interface Scenario t	
	deposit_cheque(CT)	Cheque.deposit	Interface Scenario t	
	mini_statement(AT)	BA.print_mini_stat	Interface Scenario t	
	ask_transfer(CT)	BA.withdraw BA.deposit OB.send_transfer	Interface Scenario t	
	set_standing_order(CT)	SO.create	Interface Scenario t	
	cancel_standing_order(CT)	SO.cancel	Interface Scenario t	
Other_Bank (OB)	receive_transfer	BA.deposit	Interface Scenario ob1	
	send_transfer		Teller(CT), SO	ob2
	cheque_withdraw		Cheque	ob3
	remote_withdraw	BA.withdraw	Interface Scenario ob1	
Standing_Order (SO)	create		Teller(CT)	so
	cancel		Teller(CT)	so
	debit	BA.withdraw BA.deposit OB.send_transfer	internal	
Cheque	withdraw	BA.withdraw	Teller(CT)	c
	deposit	BA.withdraw BA.deposit BA.perhaps_deposit OB.cheque_withdraw BA.full_deposit	Teller(CT)	c
Bank_Account (A)	create(A)		Teller(CT)	ba
[A + CA + SA]	remove(A)		Teller(CT)	ba
	deposit(A)		Teller(CT), Cheque, OB, SO	ba
	withdraw(A)		Teller(ES,CT), Cheque, OB, SO	ba
	get_balance(A)		Teller(CT)	ba
	print_mini_stat(CA)		Teller(AT)	ba
	perhaps_deposit(CA)		Cheque	ba
	full_deposit(CA)		Cheque	ba
	credit_interest(SA)		internal	
	update_date(SA)		internal	

Table 6.2: OCT with gates

The initial OCD built by following the above rules is depicted in Figure 6.8.

4. Deal with superclasses.
   In general, we do not require instances of the superclass. If this is the case remove the corresponding node from the OCD. (Coad and Yourdon have different notations for classes without instances, i.e. abstract classes, and classes with instances, but others do not [CY91a].)

5. Add more arcs.
   During the specification of each individual object we may identify new message connections, which will require more arcs in the OCD. If this happens, proceed according to steps 2 and 3.

In the later iterations object generators and new groupings are added to the model. This requires the following changes in the diagram:

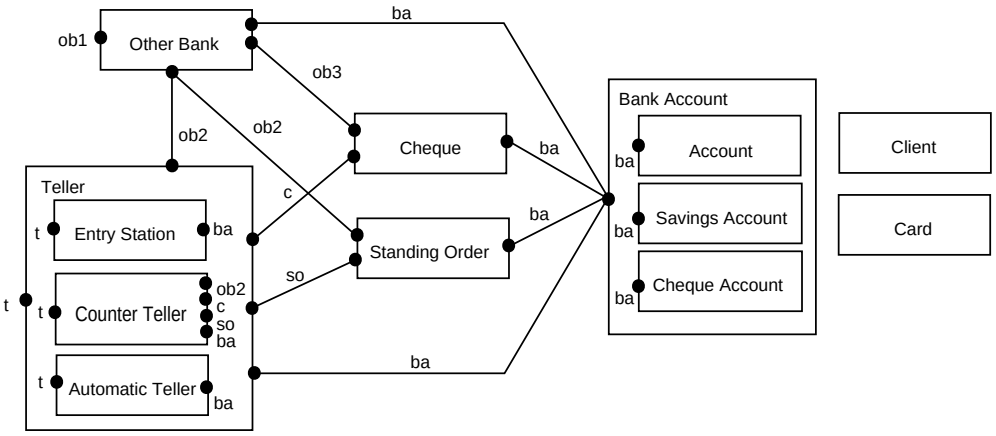

Figure 6.8: Initial object communication diagram

6. Introduce object generators.
   After introducing object generators, each node represents multiple objects, i.e. a class of objects. We use names in the singular for the class templates in the object model and for the nodes in the initial OCD, but we use names in the plural for object generators. Hence each node corresponding to a class template with a generator — not all the class templates need a generator — has a plural form of name. (The final OCD for the banking system is illustrated in Figure 6.9.)

7. Introduce new groupings of objects.
   New groupings of objects may lead us to merge some arcs. If two or more objects are grouped, we can decide to use the same communication arc for the services the component objects offer, i.e. all the services offered (not required) by the composite object or subsystem would be offered at the same gate. The idea is to treat the higher level object as one object, instead of dealing with each object component separately. When we create groupings, name clashes can occur in the OCT. To avoid this problem, we put the name of the component between brackets after the name of the service in column two (see Tables 6.2 and 6.3.) Notice, however, that name clashes do not occur in the LOTOS model as the services will be distinguished by the object identifier of the object offering the service. Therefore, using the component name between brackets is a precaution we only have to take during the construction of the object communication table.

   After this we have to change column one of the OCT to deal with the groupings and change column four to replace the name of a component with the name of its subsystem or aggregate. Next, we re-apply again the rules in step 3 of Task 3.1 to all the cases where the groupings or their components are referred. This can cause some of the gates to be amalgamated.

Task 3.2: Specify Class Templates

As our goal is to produce a formal model in LOTOS, we should not spend too much time in Tasks 2.3 and 3.1 during the first iteration. Although producing a hierarchical architecture is fundamental to the understandability of a system, and a behaviour expression with a

large number of processes may not be easily understood, we start building the formal model by modelling individual objects in LOTOS, before we have defined much hierarchy in the system. By specifying objects in LOTOS we gain more knowledge about the system and this will help in later iterations to find suitable groupings.

In Chapter 3 we discussed how to model objects, class templates, classes and inheritance in LOTOS and in Chapter 4 we discussed how to model aggregates. We now start specifying the more complex objects. The behaviour of an object is specified by a class template and its state information by one or more ADTs given as parameters of the template. And so, for each individual object, we:

1. Specify the class template by a LOTOS process definition.

2. Specify ADTs using ACT ONE.

A process definition with its ADTs fully specifies an object, i.e. defines the class template. It defines the static, dynamic and functional aspects of an object. The object model defines each class template by its name, a list of services and a list of attributes. An ADT alone could define and formalise the information. The behaviour of each class template is not, however, given by the object model. The OOA methods give it in a separate model, the dynamic model. A process [1], together with ADTs, is required formally to specify the *behaviour* of each individual object. The transformation of data and information exchanged during message passing is given by another separate model by the OOA methods, the functional model. Here, with processes and ADTs we specify the information passed during synchronization and specify the transformations which cause an object's state to change. We also model the interface scenarios as processes and each static relationship in the object model. Finally, following the structure in the OCD, we compose the processes into behaviour expressions, defining in this way the behaviour of the whole system.

We end up with an executable specification where we can use rapid prototyping to validate the specification against the requirements.

During the whole process we may find objects which we decide to specify as a single ADT. These objects usually play a secondary role in the system, acting only as attributes of other objects. However, they can be promoted, to be specified with a process, if we change their importance in the problem.

Task 3.2.1: Specify Dynamic Behaviour

To specify the dynamic behaviour of a class template we take as a starting point columns two and three of the OCT. Column two gives the services offered by the class template and column three gives the services required by that class template, from another class template. Then, we have to find out the order in which the calls to the required services occur. For that, we should imagine ourselves inside an object specified by the class template and act as if it was the centre of the system. The offered services are often represented in the process as the alternatives in a choice expression. The required services usually come in sequence after an offered service in a behaviour expression. In LOTOS, objects communicate via event synchronization which takes place through gates. The arcs in the OCD correspond to gates through which the objects communicate.

[1] An object corresponds to the instantiation of a LOTOS process. When the distinction is clear from the context, we use the term *process* for both a process definition and a process instantiation.

The services offered and required are modelled in LOTOS as structured events, as discussed in Section 3.3.8. Specifying a class template as a process, we show the services and their order, we show message passing with information being passed during synchronization, and we give it a precise meaning. The ADTs given as parameters of the process give the object's state information. It is important to define the signature of the services and to describe the class template in terms of its attributes. Comparing the class template defined with a process and ADTs with the class template as it appears in the object model we can appreciate how much more new information we now have. Some of this information, such as dynamic behaviour, is described by the OOA methods in supplementary models (dynamic and functional). Here, we have a single integrated model that defines the static, dynamic and functional properties in a formal manner.

In order to specify a process incrementally, only a subset of services may be dealt with in the first iteration.

For example, let us consider the class template `Cheque`. While specifying it, we discovered one more service `enquire cheque` which is only offered after a deposit or a withdraw have been offered. One solution for this could be the use of a boolean flag which, when "true", only allows a deposit or a withdrawal to be accepted. After a deposit or a withdrawal occurs, the flag will be set to "false" and only state information enquires are then allowed. The alternative, which we prefer, is to split the cheque process into two processes, one dealing with the active part of the cheques (deposit and withdraw operations), and other dealing with the passive part (enquiries about the cheque state information). This approach is related to the discussion in Chapter 3, Section 3.3.7, about the state of an object. Once more, we choose the solution which gives us less code. So, we specify `Cheque` by using the two auxiliary processes: `Active Cheque` and `Passive Cheque`.

```
process Cheque[c, ob3, ba](ch: Cheque_Number) : noexit :=
  ( Active_Cheque[c, ob3, ba](ch)
   >> accept this_cheque: State_Cheque in Passive_Cheque[c](this_cheque)
  )
endproc
```

First, we would like to mention that the parameter defined in `Cheque` is not exactly the kind of sort we would expect. This was the first solution we thought of to deal with cheques. To start with, we only define a simple ADT which is composed of the attribute `Cheque_Number`. Only then does the template `Passive Cheque` deal with the full state of cheques. Secondly, notice that we cannot access `Passive Cheque` until `Active Cheque` is finished. This sequentiality is introduced by the enable operator `>>`.

`Active_Cheque` is then specified as follows:

```
process Active_Cheque[c, ob3, ba](ch: Cheque_Number) : exit(State_Cheque) :=
  ( ( c  !deposit ?id: Id_Tellers ?bk: Bank_Name
         ?ch_from: Account_Number ?ch_to: Account_Number !ch ?m: Money;
      ba !perhaps_deposit !ch_to !m;
      ( [bk eq This_Bank] ->
        ( ba !withdraw !ch_from !m;
          ba !rtn_withdraw !ch_from ?cheque_valid: Bool;
          ba !full_deposit !ch_to !m !cheque_valid;
          exit(Make_Cheque(ch, ch_from, ch_to, bk, m))
        )
```

```
        []
         [bk ne This_Bank] ->
         ( ob3 !cheque_withdraw !bk !ch_from !ch !m; (* to other banks *)
           ob3 !rtn_cheque_withdraw !bk !ch_from !ch !m ?cheque_valid: Bool;
           ba !full_deposit !ch_to !m !cheque_valid;
           exit(Make_Cheque(ch, ch_from, ch_to, bk, m))
    ) ) )
    []
     ( c !withdraw ?id: Id_Tellers ?bk: Bank_Name
         ?n: Account_Number !ch ?m: Money; ... (* with my cheque in my bank *)
  ) )
endproc
```

When an instance of `Active Cheque` asks for the service `full deposit` from an instance of `Cheque_Account`, the behaviour of `Cheque Account` depends upon the value of the boolean variable `cheque_valid`. If this variable takes the value 'true', the amount `m` is added to the current balance, otherwise it is not.

Finally, `Passive_Cheque` can be:

```
process Passive_Cheque[c](this_cheque: State_Cheque) : noexit :=
  ( c !enquire_cheque ?id: Id_Tellers
      !Get_Cheque_Number(this_cheque) !this_cheque;
    exit(this_cheque)
  ) >> accept update_cheque: State_Cheque in Passive_Cheque[c](update_cheque)
endproc
```

Having discovered one more service (`enquire_cheque`), we have to update the OCTs and the refined object model accordingly.

Task 3.2.2: Specify ADTs

An ADT defines the necessary equations to allow the objects to be prototyped with state information and values to be passed during the communication, but without giving too much detail about how each service is performed internally. The ADT describes the attributes of an object and the operations which deal with those attributes.

As with a process, an ADT can be specified incrementally. We can start by specifying only some of the attributes and services and add more detail in later iterations.

In Chapter 3 we discussed how to model ADTs in LOTOS and we gave the rules to be followed when applying ROOA. As the ADT we presented there for an account is more interesting than the ADT where the sort `State Cheque` is defined, we do not show the LOTOS code for `Cheque_Type`.

The order in which we specify processes and ADTs (Tasks 3.2.1 and 3.2.2) is arbitrary. We can start with a process and then move to the corresponding ADTs, or we can start with the ADTs and then move to the process. We may decide to start specifying a group of processes and then specify the ADTs, or vice-versa. Also, part of the system may be fully dealt with, ignoring the rest.

The application of this task (Task 3.2) may require application of Task 2.2, i.e. if further relationships, attributes, services and message connections are identified, they must be added to the refined object model. Task 3.1 must also be applied if new message connections are identified.

Task 3.3: Compose the Objects into a Behaviour Expression

Following the structure of the OCD, we compose the objects defined in Task 3.2 into a LOTOS behaviour expression by using the parallel operators. The algorithm by Clark [Cla94b] describes how this can be done for an OCD of arbitrary complexity and identifies situations in which an OCD cannot be represented in LOTOS. The main point of this algorithm is that a server cannot be grouped with a subset of its clients.

Notice that, since in the first iteration each node in the OCD represents a single object, the composed behaviour expression is built of single objects. For example, the objects that form the OCD in Figure 6.8 would be composed as follows:

```
( Other_Bank[ob1, ob2, ob3, ba](Make_Bank(...))
 |[ob2, ob3]|
 ( Teller[t, ob2, c, so, ba]
  |[c, so]|
  ( Cheque[ob3, c, ba](Make_Cheque(...))
  |||
    Standing_Order[ob2, so, ba](Make_SO(...))
 ))
|[ba]|
 Bank_Account[ba]
)
where ...
```

Before we can start the next task, we have to model the interface scenarios and compose them, one at a time, with the behaviour expression above. Consider the following part of an interface scenario for the banking system:

```
process Interface_Scenario[t, ob1]: noexit :=
  (* create an automatic teller and a counter teller *)
    t !create ?idc: Id_Tellers;
    t !create ?ida: Id_Tellers;
  (* open a cheque account from counter teller *)
    t !open_account !idc !cheque;
    t !rtn_open_account !idc ?nc: Account_Number;
  (* deposit from counter teller *)
    t !deposit_cash !idc !nc !This_Mon;
    t !rtn_deposit_cash !idc !nc !This_Mon;
  (* balance from automatic teller*)
    t !print_mini_statement !ida !nc;
    t !rtn_print_mini_statement !ida !nc ?a: Account;
  (* withdrawal from automatic teller *)
    t !withdraw_cash !ida !nc !This_Mon;
    t !rtn_withdraw_cash !ida ?val: Bool;
    ...
   (hide success in success; stop)
endproc
```

The `Interface_Scenario` process acts as if it was the client of the whole system. It initiates calls to the tellers on gate `t` and to other banks on gate `ob1`, and waits for the respective answers. We composed it in parallel with the above behaviour expression, by using gates `t` and `ob1`, as follows:

```
( ( Other_Bank[ob1, ob2, ob3, ba](Make_Bank(...))
    ...
  )
 |[ba]|
  Bank_Account[ba]
)
|[t, ob1]|
 Interface_Scenario[t, ob1]
```

In later iterations, when object generators are introduced to deal with multiple instances, the composed behaviour expression is refined and built up of a combination of object generators and of single objects (in cases where generators are not required).

We may decide to deal only with part of the system and then, in further iterations, add more objects until the whole system is considered.

Task 3.4: Prototype the Specification

We use interface scenarios and rapid prototyping to check services, message connections and attributes. The syntax and static semantics of the LOTOS specification are checked by the LOTOS tools and the specification can be prototyped by using SMILE or some other LOTOS simulator. Any errors, omissions or inconsistencies found during the simulation will lead us to iterate Tasks 2.2, 3.1, 3.2 and 3.3 and to update the original requirements document, the object model, the OCT and the OCD.

In the first iteration, as the emphasis is on ensuring that the individual class templates have been correctly specified, a behaviour expression consisting of single instances of class templates is prototyped. In later iterations, multiple instances are dealt with and we check that the complete system has been properly specified.

Task 3.5: Refine the Specification

We refine the specification by re-applying Tasks 2.3, 3.1, 3.2, 3.3 and 3.4. During successive refinements we may:

1. Model static relationships.
2. Introduce object generators.
3. Identify new higher level objects.
4. Demote an object to be specified only as an ADT.
5. Promote an object from an ADT to a process and an ADT.
6. Refine processes and ADTs by introducing more detail.

Task 3.5.1: Model Static Relationships

The first action we have to take in the second iteration is to model conceptual relationships. Notice that at this stage we are still dealing with a single instance of each class template defined in the object model.

Depending on its cardinality, a static relationship is either modelled as an attribute, or as a set of attributes, in one of the objects involved in the relation (or both if the relationship is bidirectional), as described in Chapter 3.

This task involves Tasks 3.2, 3.3 and 3.4.

Task 3.5.2: Introduce Object Generators

During a first iteration we deal only with a single instance of each class template. This simplifies the problem and allows us to prototype with a specific number of objects. However, in general, several instances of the same class may be required. This is achieved by defining an *object generator* for a class template.

When dealing with subsystems, we can decide to define an object generator for each component, or else define an object generator for the whole subsystem. Which is to be preferred depends on each particular situation.

In Chapter 3 we discussed in detail how ROOA specifies object generators in LOTOS. There we presented the most often used view of an object generator. However, to deal with certain specific problems we may require a variation of the object generator. ROOA allows it. For example, `Cheque` is one of those special cases. During later refinements of the LOTOS specification, this class template was one of the class templates that suffered more changes. One of the important changes was to consider that we had two kinds of cheques: the ones to withdraw and the ones to deposit. So, we split `Cheque` into a `Cheque_Withdraw` and `Cheque_Deposit`. Another difference in cheques is that the object identifier is not generated by the system. These lead us to define an object generator which can be used to create objects of both kinds:

```
process Cheques[c, ob3, ba] : noexit :=
  c !create_cheque ?ch: Cheque_Number ?bk: Bank_Name
    ?from_acc: Account_Number ?to_acc: Account_Number ?m: Money;
  ( Cheque_Deposit[c, ob3, ba](Make_Cheque(ch, from_acc, to_acc, bk, m))
  |||
    Cheques[c, ob3, ba]
  )
 []
  c !create_cheque ?ch: Cheque_Number ?bk: Bank_Name
    ?n: Account_Number ?m: Money;
  ( Cheque_Withdraw[c, ob3, ba](Make_Cheque(ch, n, n, bk, m))
  |||
    Cheques[c, ob3, ba]
  )
endproc
```

The object generator has no parameters now, but each branch of the choice has still the same structure of the object generators discussed in Chapter 3.

This task affects Tasks 3.1, 3.2, 3.3 and 3.4.

Task 3.5.3: Identify New Higher Level Objects

The identification of new higher level groupings (subsystems and aggregates) leads us to change both the initial OCD and the OCT in order to incorporate the new objects. Therefore we should apply again Tasks 3.1, 3.3 and 3.4.

In the banking example, we grouped `Cheques` with `Standing Orders` to form the subsystem `Financial Instruments`. These changes can be seen in the OCT represented in Table 6.3.

Class Templates	Offered Service	Required Service	Clients	Gates
Teller	open_account(CT) ...	BA.create	Interface Scenario	t
Other_Bank (OB)	receive_transfer send_transfer cheque_withdraw remote_withdraw	BA.deposit BA.withdraw	Interface Scenario Teller(CT), FI(SO) FI(Cheque) Interface Scenario	ob1 ob2 ob2 ob1
Financial_Instrument (FI) [SO + Cheque]	create(SO) cancel(SO) debit(SO) withdraw(Cheque) deposit(Cheque) ...	 BA.withdraw BA.deposit OB.send_transfer BA.withdraw BA.withdraw BA.deposit BA.perhaps_deposit OB.cheque_withdraw BA.full_deposit	Teller(CT) Teller(CT) internal Teller(CT) Teller(CT)	cs cs cs cs
Bank_Account (BA)	create remove deposit withdraw get_balance ...		Teller(CT) Teller(CT) Teller(AT,CT), OB, FI(Cheque, SO) Teller(ES,CT), OB, FI(Cheque, SO) Teller(CT)	ba ba ba ba ba

Table 6.3: Refined OCT

The revised OCD, based on Table 6.3 and after we have introduced object generators, is depicted in Figure 6.9.

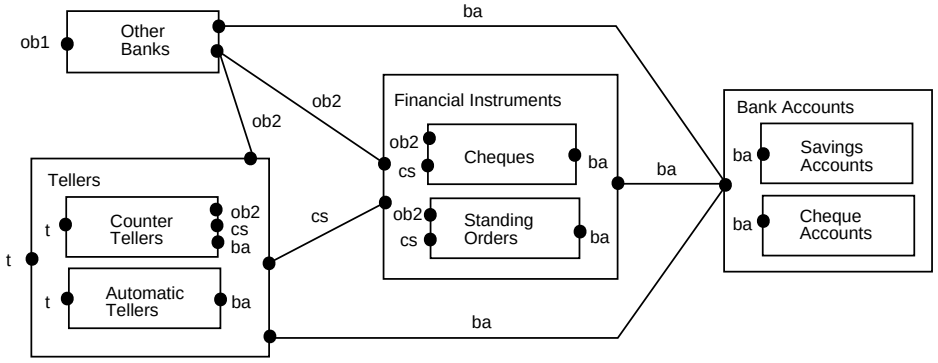

Figure 6.9: Revised object communication diagram

Now, the composition of the object generators in the OCD would take the form:

```
( Other_Banks[ob1, ob2, ba](Insert(This_Bank, {} of Bank_Name_Set))
|[ob2]|
 ( Tellers[t, ob2, cs, ba]
 |[cs]|
```

```
    Financial_Instruments[ob2, cs, ba]
))
|[ba]|
 Bank_Accounts[ba]
```

Task 3.5.4: Demote an Object to be Specified only as an ADT

If an object plays a secondary role in the system, i.e. it only acts as an attribute of other objects, it should be specified as a single ADT.

In this case, delete that object from the OCD. This affects Task 3.1, 3.2, 3.3 and 3.4. Note that in Task 3.2 we only need to delete the process corresponding to that object.

Task 3.5.5: Promote an Object to be Specified as a Process

An object that we considered to have a secondary role in the system may rise in importance when we add more detail to the specification. Because we allow processes and ADTs to be specified incrementally, new information can have this effect on the formal model.

This task affects Task 3.1, 3.2, 3.3 and 3.4.

Task 3.5.6: Refine Processes and ADTs

The complete definition of a process or an ADT can be done incrementally. In each refinement we can add more detail to the specification. When more information is added to the formal model, more static relationships, attributes, services, and message connections can be identified. In this case, add them all to the object model and apply again Tasks 2 and 3.

In the specifications we developed by using ROOA, there were major differences between the initial ones and the final ones. These differences are of two kinds: structure and detail. The structure of a specification can change according to the extra complex objects and subsystems we find during refinement. The detail involved when specifying each class template for the first time depends on the experience already acquired with ROOA. Even then, we advise complex objects to be specified incrementally. Start by just using the services (and attributes) we understand about, leaving for further iterations the ones which we do not understand well. A good example of this was the class template `Cheque`. It kept changing during the application of the method. The final version of the process specifying `Cheque_Deposit` is as follows:

```
process Cheque_Deposit[cs, ob2, ba](this_cheque: Cheque) : noexit :=
  ( Cheque_Deposit_1[cs, ob2, ba](this_cheque)
   >> accept this_cheque: Cheque in Passive_Cheque_Deposit[cs](this_cheque)
  )
  where
    process Cheque_Deposit_1[cs, ob2, ba]
              (this_cheque: Cheque) : exit(Cheque) :=
      ba !perhaps_deposit !Get_Acc_To(this_cheque)
         !Get_Amount(this_cheque);
      ( [Get_Bank(this_cheque) eq This_Bank] ->
        ( ba !withdraw !Get_Acc_From(this_cheque) !Get_Amount(this_cheque);
          ba !rtn_withdraw_ok !Get_Acc_From(this_cheque) ?cheque_valid: Bool;
          ba !full_deposit !Get_Acc_To(this_cheque)
```

```
                !Get_Amount(this_cheque) !cheque_valid;
              exit(this_cheque)
            )
           []
            [Get_Bank(this_cheque) ne This_Bank] ->
            ( ob2 !cheque_withdraw !Get_Bank(this_cheque)
                  !Get_Acc_From(this_cheque) !Get_Cheque_Number(this_cheque)
                  !Get_Amount(this_cheque);       (* to other banks *)
              ...
          ) )
        endproc (* Cheque_Deposit_1 *)

        process Passive_Cheque_Deposit[cs](this_cheque: Cheque) ... endproc
    endproc (* Cheque_Deposit *)
```

With the refinement of the template we had to refine also the ADT `Cheque Type` so that operations such as `Get Amount` were defined.

There is not a clear boundary between analysis and design; there never was. Therefore the old question "when does analysis finish and design start?" is still an open question. However, before we move to the design, we have to ensure that the requirements specification is internally consistent and deals with all the essential objects identified from the original requirements. For a specification to be internally consistent, we have to guarantee that, for every message connection, there are appropriate events in the calling and the called objects, for every static relationship there are all the objects involved in the relationship and a complete trace through the system can be made for every interface scenario.

6.4 The ROOA Documents

The most useful form of describing a process is in terms of work products [PC86]. ROOA is not only a process of developing software. It also produces documentation as the process is applied. In Figure 6.10 we show the products built by ROOA.

The object model is produced by Task 1 and the information it contains depends on the object-oriented analysis method used. The refined object model is produced by Task 2 and it includes an object model where the objects are described with a list of attributes and a list of services. This object model also describes the static and dynamic relationships between objects and the interface objects (when necessary). During Task 2 we also define interface scenarios to model the interaction of the system with its environment. The OCT (object communication table) is a table developed during Tasks 2.2.2, 2.2.3 and 3.1. It helps to define the services offered and required by each object, the message connections between objects and the points of synchronization between the objects. The OCD (object communication diagram) is a graph that represents the dynamic structure of the final LOTOS specification. Finally, the LOTOS specification is developed from Task 3.2 to Task 3.5.6.

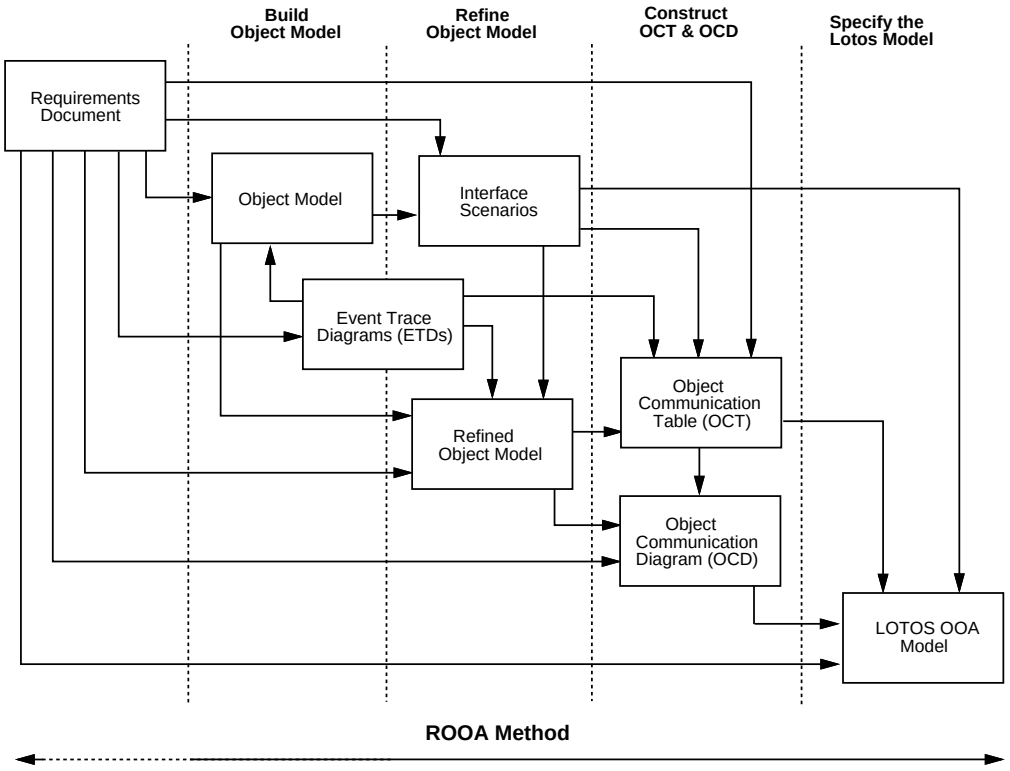

Figure 6.10: Documents produced during the application of ROOA

6.5 Conclusions

This chapter together with Chapters 3 and 4 describes the final product of our investigation, i.e. the Rigorous Object-Oriented Analysis (ROOA) method. Therefore, Chapters 3 and 4 are an important part of the ROOA method as they show how each object-oriented concept is modelled in LOTOS.

The ROOA method integrates LOTOS with existing OOA methods, producing a formal object-oriented analysis model which acts as a requirements specification of the system. It proposes a systematic development process which is composed of three main tasks: build an object model; refine the object model and identify dynamic behaviour; build a LOTOS formal object-oriented analysis model. We have discussed each of these tasks in detail by discussing each of their subtasks which are also composed of subsubtasks.

The final ROOA model integrates the static, dynamic and functional properties of a system, unlike other OOA methods which produce three separate models which are difficult to integrate and keep consistent. This model is primarily a dynamic model, but it maintains the structure of the static object model. This is very important since human beings find static models easier to understand.

A major task in the creation of a LOTOS specification is the transformation from a static to a dynamic model. The OCT and the OCD are two intermediate structures used to help in this transformation.

ROOA uses a stepwise refinement approach for the development and for validation of the specification against the requirements. The development process is iterative, allowing

us to apply ROOA to objects at different levels of abstraction and to refine the model incrementally.

As the final model is formal and executable, ROOA uses prototyping to detect and correct errors, inconsistencies, ambiguities, etc. early and to give feedback to the requirements capture.

Chapter 7

The Design Rationale of ROOA

7.1 History

We initiated the ROOA project in January 1992. It took us over two years to reach the state in which it is now. During this time, ROOA's development benefited from: being applied to different problems (including its application by other people), and continually to the banking system; regular current literature surveys in the object-oriented field; comments of reviewers and colleagues on our publications. The books by Rumbaugh *et al.* [RBP+91] and the one by Jacobson [Jac92] influenced our work a lot and helped us considerably in developing ROOA to its current form. Until these appeared, the existing OOA methods emphasised the object model, as happened with Coad and Yourdon's method [CY91a]. OMT [RBP +91] and OOSE [Jac92] propose additional complementary models which helped us to feel confident about the role of the LOTOS model we were producing.

Initially, our approach to building the LOTOS model was mainly to provide a specification which described the external behaviour. Later, ROOA consisted of formalising the object model, adding to it dynamic and functional properties which were not contained in the purely static architectural object models. To formalise the object model, we had to devise a method for specifying in LOTOS each concept appearing in an object model. We began by using the OOA method by Coad and Yourdon [CY91a]. This method is based upon the concepts of *class*, *object*, *relationship*, *message passing* between objects, *inheritance* and *aggregation*. Mapping these concepts into LOTOS is an important part of the ROOA method. The LOTOS model produced by ROOA is primarily a dynamic model and is therefore good at the description of behaviour. Nevertheless, its structure is directly related to the structure of the object model and so it is equally good at describing the static properties.

ROOA, as described in this thesis, is a result of successive refinements and the effort we made to encapsulate within ROOA the diverse object models created by most of the OOA methods. During ROOA development it was important to keep in mind that as we were producing an analysis method, we should not engage in decisions that are better left to the design phase. Consequently, we use non-determinism and value generation heavily, and we employ an abstract way to specify ADTs.

This chapter explains the problems we faced during the development of ROOA. It discusses the design decisions we made and explains why we rejected some solutions in

favour of others. Section 7.2 discusses the most significant problems we had to solve. Section 7.3 presents the three main stages which ROOA passed through before it attained the final stage presented in Chapter 6.

7.2 Major Problems and Their Resolution

Parallelism. A significant decision was to model objects as processes instead of having a process controlling a list of objects, each one specified as an ADT. Even passive objects are created by means of object generators. Consequently, we have sets of processes instead of one process controlling a set of ADTs.

Our first solution of the banking system exercise modelled each account object as a value of an ADT. All accounts were specified with the process `Data Base` which has the argument `DB_Accounts_Type`. `DB_Accounts_Type` is an ADT which specifies a set of accounts. The process `Transactions`, running in parallel with `Data Base`, accepts the services required from the clients and passes them on to `Data Base`. This did not sufficiently abstract from the implementation.

We therefore decided to define each object as a process instance, taking advantage of the LOTOS parallel operators, and specifying a system as a set of concurrent objects. In this way we avoid premature design or implementation decisions, such as protection techniques for the concurrent access of shared data.

Classes and Objects. After deciding that ROOA should model a system as a set of concurrent objects, we started by modelling the concept of class in LOTOS as a process definition and the concept of object (of a class) as a process instantiation. Since an object is a member of a class, the object-oriented-analysis relation of class to object seemed similar enough to the LOTOS relation of process to process instantiation. However, with this mechanism, the LOTOS versions of objects could only be created statically, at specification time. This was a critical failing. An object model does not specify, in general, how many objects of each class are actually instantiated in a running system. Also, we felt, this is not part of the requirements document in general. Hence we needed a mechanism for specifying, in LOTOS, classes and objects without committing ourselves to a specific number of objects.

A Generic 'Object-Generator' in LOTOS. We wanted to be able to create objects dynamically, without *a priori* limitation on the number of possible objects.

This led us to devise the concept of *object generator*. The idea of dynamically creating processes was already used in LOTOS, by using the interleaved operator `|||` and recursivity. This is the basic structure of an object generator. However, an object generator uses a service with a selection predicate to explicitly create the objects and uses value generation of object identifiers to hold information about the objects already created. Object generators allow us to instantiate a variable number of objects during the simulation of a specification. The ROOA specification thus contains no limitation on the number of objects allowed. Nevertheless, the LOTOS specification may be tested by symbolic execution.

The Implementation of an Object Generator. Initially, we generated natural number values for use as object identifiers. This is a cumbersome procedure in LOTOS. LOTOS data types are algebraic data types with few facilities to make them easy for developers to use. For example, the naturals consist of the terms 0, $succ(0)$, $succ(succ(0))$,..., but we would have preferred to use the normal symbols 0, 1, 2, 3, etc.

During the development of ROOA, we had three options to deal with the object identifiers. First, we thought that the user (here symbolised by the interface scenario) would give to the object generator a value to be used as the object identifier. This seemed to us a way to avoid the problem, without solving it. Next, we defined a special function to generate natural numbers as object identifiers. We were never happy with this solution as it seemed to be too close to an implementation decision. As the LOTOS simulator SMILE supports value generation, we found that this technique was preferable to creating arbitrary object identifiers. So, for the third solution, we started using value generation to create the identifiers.

Classes and ODP. In the OOA literature [CY91a, Jac92, RBP $^{+}$91], the concept of class appears with two mutually incompatible definitions:

- a class specifies the common characteristics of objects of the same kind;
- a class is a set of objects of the same kind.

To choose which definition to follow we took the definitions from the ODP model [ISO94], which proposes *class templates*, *classes* and *objects*. A class template is specified by a process definition, an object is a process instantiation and a class is the set of objects instantiated from the same class template. A class has some similarities with our notion of an object generator, since the object generator holds the set of object identifiers of the objects which have already been instantiated.

Refining the Notion of Class Template. During the development of a class template as a process definition in LOTOS, we had to make some decisions concerning the information it should contain. In LOTOS, when we created an instance of a class template, we could not give a name to the process instance being created. This led us to formulate the need for *object identifiers*. An object identifier is an attribute of an object with a similar function to that of a *key* in a relational database. A second object uses the object identifier to specify which object it is to communicate with.

State. The state of an object at any time is given by the configuration of the values of the attributes of the object at that time. This is handled in LOTOS by defining the (type of the) attributes as one or more ADTs which are then given as parameters of the process defining the class template.

As LOTOS gives few built-in facilities, specifying full-blown ADTs may be a complex and lengthy task. Each operation has to be defined by one or more (probably many) equations. However, as we are in the analysis phase, we did not want to give too much detail about how services are handled internally. Therefore, we thought that we could get by using simpler ADTs with a small number of equations. Our first solution was to use *dummy ADTs*. In a dummy ADT, modifiers return the initial value and selectors return

always a symbolic constant value. For example, in the banking system, the equation specifying a credit into account `acc` with amount `m` is defined as

```
Credit_Account(acc, m) = acc;
```

and getting the balance of that account would be given by the equation

```
Get_Balance(acc) = Some_Balance
```

where `Some_Balance` is a constant value defined in an ADT. The value `acc` is of sort `State_Account` defined in an ADT and it represents the state of an account. These equations give abstract state information to the objects, define the signature of the operations and indicate the type of the information passed when objects communicate. However, they do not allow us to record any information about the event history of the objects, for example.

As in some types of systems a record of the history of events, for example an 'audit trail', may be required, we had to reject the above solution and propose another. For the new solution, we changed the specification of the ADTs (which we then called *Symbolic ADTs*) according to their operations. Modifiers are specified without equations. This enables us to treat them as ADT constructors, and so we gain a record of the history of events. Selectors for which a particular result does not need to be returned are defined with dummy equations. Selectors which need to return a particular value are defined with one equation for each constructor.

The use of the generalised **choice** operator in the process part allows us to cover all the possible states which are not covered in the ADTs, as shown in Chapter 3.

Communications. Work on communication between objects in LOTOS was already available at the time of initial development of ROOA [Cla92b]. Besides, it is straightforward to define communication between objects using LOTOS constructs, such as *synchronisation* in *gates*. Events defined in processes synchronise in gates.

We started by defining the structure of a LOTOS event. Initially, the specification produced by the first iteration of the ROOA method was written in a form equivalent to basic LOTOS. An initial version of a structured event in LOTOS had the form:

⟨*gate name*⟩ ⟨*message name*⟩

where *message name* corresponds to the service offered by the object. With the introduction of ADTs and the refinement of ROOA, the structured event became:

⟨*gate name*⟩ ⟨*message name*⟩ ⟨*caller object id*⟩ ⟨*called object id*⟩ ⟨*parameters*⟩ ⟨*result*⟩

As communication may be non-atomic, for example as in a remote procedure call or an Ada rendezvous, we thought that the structured event should include the identifier of both objects involved in the synchronisation. This solution did not survive for long, since to model remote procedure call or rendezvous an object needs to know the identifier of the object it is to communicate with, but the second object does not need statically to know the identifier of the object that called it. Finally, the structure of the LOTOS event took the form:

⟨*gate name*⟩ ⟨*message name*⟩ ⟨*called object identifier*⟩ ⟨*parameters*⟩

this reflects the structure of both the client's call and the server's answer.

The Object Communication Diagram (OCD). How to define the number of gates necessary in the specification turned out to be one of the more difficult problems we had to solve. The OCD appeared in the early stages of the ROOA development, mainly to help us solve this problem.

Initially, we believed that all the services offered by a class template should be on offer on a single gate, but we soon discovered that this solution would not work (see Figure 7.1). This could be because a gate is, by definition, to be used for communication between two (or more) processes and not to be thought of as the private property of a single process.

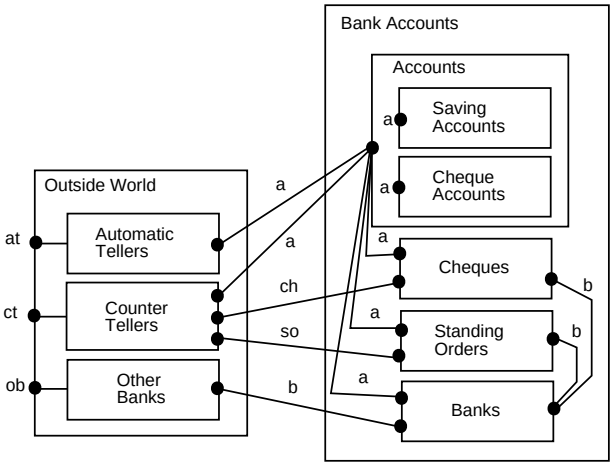

Figure 7.1: Class templates offering all their services in a single gate

We then decided to assign gates according to the structure of the specification. This second solution would assign one gate for communication between two subsystems, between two complex objects or between one subsystem and one complex object, whose structure we already knew, as in Figure 7.2. This way we reduced the number of gates, but we ended up with another problem: a class template could have to offer the same service at different gates. Therefore, we required a more satisfactory solution.

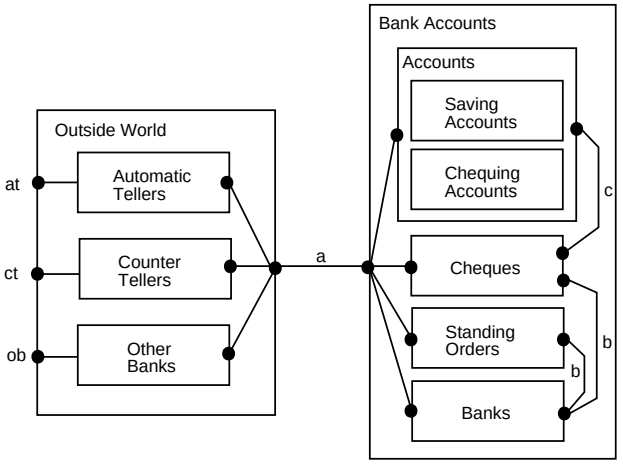

Figure 7.2: Giving gates according to the structure

The third attempt defined gates according to the types of objects and required that an object could not use the same gate to communicate with both an object at the same level of abstraction and another object at a different level of abstraction. We then formulated the following principles for assigning gates:

1. A server which is a passive history-sensitive object offers its services at a single gate.

2. Two active objects at the same level of abstraction communicate through a single gate.

3. Objects which make up a composite object may only interact with each other through internal gates.

At this stage we were classifying objects according to their role. Using this approach as a basis (it changed, as we will see below), and after several refinements (including the introduction of the OCTs) we end up with the OCD depicted in Figure 7.3. (In this diagram we merged the object `Banks` within `Other Banks`.)

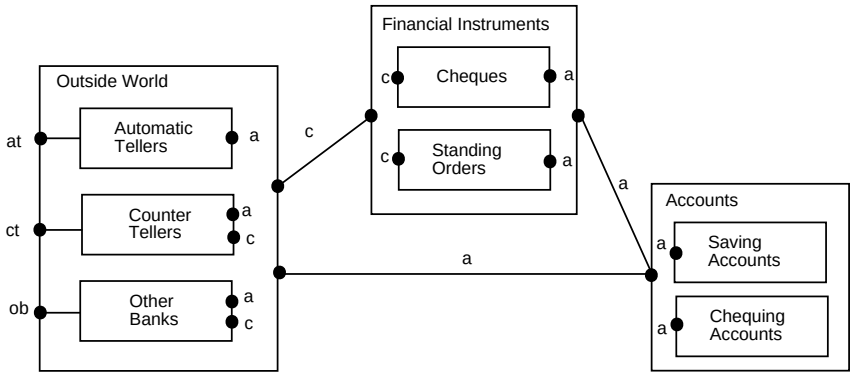

Figure 7.3: Process `Accounts` is only accessed by a single gate

With the introduction of the final rules, as shown in Chapter 6, the diagram also changed to that presented in Figure 6.9.

The OCD proved to be very useful, since it helps in the transformation from a static to a dynamic model. It can be built directly from the object model and the OCT, and it directly models the structure of the LOTOS specification which is represented by the top level behaviour expression.

Classifying Objects. For some time, in order to define the number of gates, we thought that classifying the objects according to their role in the system could help determine their specification in LOTOS and the necessary gates. Our first attempt yielded *passive* and *active* objects, but after some time, we divided passive objects into *history-sensitive* and *pure ADTs*. History-sensitive objects are those which can remember their previous local state history. Pure ADTs are those which do not have state, i.e. they are values.

After several refinements, we gave this up. The role an object played in the system in terms of having or not having significant behaviour was most important. We also discovered that it would be important to classify objects into *clients* and *servers*.

Objects-as-Processes versus Objects-as-ADTs. While working through the first large example with the ROOA method, we realised that some class templates in the object model should be specified with processes and ADTs, but others could only be specified as ADTs (one ADT for each class template). Which method is chosen should depend on the relative significance of the behaviour of an object of a given class template. For example, suppose that an object is only required to play the passive role of an attribute of another object. Such an object has no dynamic role to play in the system, and may then be specified as an ADT. On the other hand, an object with complex behaviour must be specified as a LOTOS process with one or more ADTs defining its state. Between these two extremes, the analyst must judge as to which method is preferable in the circumstances.

The Role of Interface Scenarios. After having specified all the class templates, it is necessary to 'assemble' them in some fashion in order to prototype the system by simulation (using symbolic execution, as in the LITE simulator [EW93]). To drive the simulation of the specification we used *interface scenarios* (first called *test scenarios*), which represent typical dialogues between the system and the user. (They ask services from the specification and record the results.) The interface scenarios initially appeared simply to help in prototyping the specification, by narrowing the options available. Limiting the event offers during the simulation is essential so that the developer does not get lost in tracing large numbers of alternative paths. Soon the role of interface scenarios expanded when we realised that they could be used to help us understand the expected dynamic behaviour of the system, and to identify services and communication between objects in the object model.

The *use cases* introduced by [Jac92] influenced our work. Our interface scenarios are not textual descriptions of the behaviour of the system, as Jacobson's use cases are, but when joined with the object model in the beginning of the development and later joined with the LOTOS specification they provide the same information that use cases give.

Refining the Method by Refining the Object Model. As we were developing ROOA, many authors came up with new methods for object-oriented analysis. We started working with OOA [CY91a], but then OMT [RBP $^{+}$91] and OOSE [Jac92] appeared. The object models produced by each method differed in the amount of information shown. We realised that ROOA should have the flexibility to incorporate each different method, depending on the user's preference. This flexibility should also be encapsulated in a task so that the core of ROOA remained the same. Thus we introduced the task: *refine the object model*. ROOA was then composed of three main tasks as presented in Chapter 6.

The refinement of the object model takes an object model obtained by applying any of the existing OOA methods, and transforms it to include all the information required as input to the task of building the LOTOS formal model.

Integrating Static, Dynamic and Functional Properties. Initially, we tried to start from an object model and write a LOTOS specification which would include both the static properties defined in the object model and the dynamic and functional characteristics from a requirements document. Since this was attempted as the specification was being written, we soon discovered that this approach was much too difficult to accomplish all at once. Instead, we decided to use intermediate structuring techniques which would

simplify the task of building the LOTOS specification. These structuring techniques are the *Object Communication Diagrams*, the *Object Communication Tables* and the *Event Trace Diagrams*.

The Object Communication Table (OCT). As more examples were attempted, other necessities appeared. For example, message connections in the object model only show that one object needs to communicate with some other object, but not the structure of this communication. Thus the OCT was developed as an intermediary structure between the object model and both the OCD and the LOTOS specification. The OCT became useful for helping to identify the services offered by each class template, and message connections. Later, with the refinement of the rules defining the number of gates, we added to it a fifth column to show the gates needed. With the incorporation of OCTs into ROOA, the role of the interface scenarios expanded further. The interface scenarios were very useful in helping us decide, when building the OCT, which services were required from the system.

Event-Trace Diagrams. Many applications of OOA methods are database-oriented problems, with a small dynamic component. Other problems are process-oriented and have a much more important behavioural component. In these problems, the types of object interactions consist mainly in a series of queries and responses. These are specified graphically by using message-passing diagrams, such as those in OOSE [Jac92]. During the application of ROOA to the warehouse problem proposed by Jacobson [Jac92], we decided to introduce as part of the method *event trace diagrams*. These diagrams are instances of Message Flow Graphs, a precise semantics for which may be found in Ladkin and Leue [LL94].

Aggregation and Subsystems. One of the design considerations of ROOA was to make it applicable to large systems. To deal with complexity, the object-oriented literature proposes, amongst other techniques, aggregates and subsystems. The distinction between these two concepts is not always made clear. Authors use different terminology to refer to the same or similar concepts, e.g. compositional object, associative object, container object. After understanding the terms used by different authors we chose ours and modelled them in LOTOS. An important decision was to decide that aggregates and their components would have proper identifiers while subsystems would merely be groupings of class templates and therefore would not have identifiers.

7.3 ROOA: Main Previous Versions

7.3.1 ROOA: First Version

In June 1992, ROOA was mainly helping us to describe the external behaviour of a system. At the time we were especially concerned with creating a method to help us use LOTOS, and we did not think of combining OOA methods with LOTOS until later. We developed the following strategy to create LOTOS specifications:

1. Look at the problem in its environment, i.e. integrate the problem into the surrounding world.

2. Identify external entities with which the system interacts.
3. Identify the kind of interactions between the system and the external entities.
4. Group the "interactions" (events) for each object and define a gate for each group. This can be done in different ways: one gate for all events; one gate for each event.
5. Think about the events and decide how they affect the system (concurrency, multi-user, mutual exclusion, etc.).
6. Detect possible internal objects.
7. Define processes. Each process in the more abstract level will implement an interface with an external object, using the defined gate.
8. Define the data structures.

The order of the steps, and the steps themselves, reflect the problems faced when trying to obtain our first LOTOS specification. And, in some way, they reflect also the decisions taken.

It is important to notice that most of this LOTOS specification was sequential, in contrast to our final versions where parallelism plays a dominant role. So, in our banking system each account was specified as an ADT. A set of accounts was then given as a parameter of the process `Data Base` which deals with each transaction a client may require from an account (see Figure 7.4).

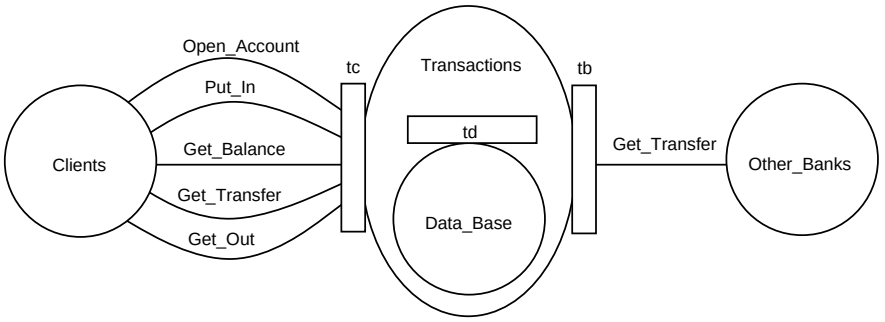

Figure 7.4: A first version of the banking system in a first version of an OCD

Each bubble represents a process, the arcs represent communication between processes and the rectangles labelled `tb`, `tc` and `td` represent gates.

7.3.2 ROOA: Second Version

From the first version to the second, the most important decision we took was to combine the OOA methods with LOTOS. To start with we thought we would have two main tasks: build an object model, and build a LOTOS model. The first OOA method we used to build the object model was OOA proposed by Coad and Yourdon [CY91a]. By December 1992, the ROOA method had developed to include the following main sequence of steps:

1. Define the external behaviour.

   (a) Identify the external entities.
   (b) Couple each external entity to one interface subsystem.
   (c) For each interface subsystem, define the services provided.
   (d) Incorporate a central subsystem that acts, to begin with, as a "data base" of the information in the system. It also acts as a medium through which the different interfaces can interact with one another.

2. Apply the Coad and Yourdon method.

3. Complement the object structure of the objects by identifying how to model the relationships in the object model: as one attribute in one of the objects, as one attribute in each object, as another object.

4. Use LOTOS to formalise the object model and to add the dynamic and the functional models.
   (a) Specify the system using the process part of LOTOS.
   (b) Incorporate a test scenario to drive the simulation.
   (c) Add dummy ADTs.
   (d) All the previous steps are validated by using LITE tools.

The first step can be considered as the "remains" of our first version. Later we eliminated this explicit step as it could be seen as part of the OOA method used. However, this step was very useful in order to highlight the importance of interface scenarios and interface objects.

After this stage we saw Jacobson's classification of objects into *entity*, *interface* and *control*. The interface objects are related to the *interface subsystem* we were proposing in step 1.b).

7.3.3 ROOA: Third Version

The third important ROOA version was ready in March 1993. In this new version we took three important decisions in relation to the previous version: task 1 should be considered part of the OOA method used; task 3 could be incorporated within task 4; add to task 4 a subtask refining the LOTOS model.

So, ROOA was now composed of the following main tasks:

1. Building an object model by using any of the existing OOA methods.

2. Building a LOTOS model.
   The goal of this step is to transform an object model into a formal LOTOS model that contains all the information normally held in the separate object, dynamic and functional models. This step has five main substeps:
   (a) Specify the behaviour of the object model.
   (b) Incorporate test scenarios to investigate object behaviour.
   (c) Add dummy abstract data types to give local state to the objects.

(d) Iterate the analysis by refining processes and abstract data types.

(e) At each stage of the development, prototype the resulting specification.

From this version to the final version many refinements occurred: we introduced another main task in order to improve the object model so we could accept an object model produced by any of the existing OOA methods; we found a better approach to deal with ADTs (see Chapter 6); we started using value generation to create object identifiers; we introduced OCTs to collect information about services and objects, and with that we refined the rules to identify gates; finally, we introduced event trace diagrams to help us describe the initial dynamic behaviour before we started to write LOTOS code.

7.4 Conclusions

This chapter described the underlying ideas behind the ROOA method. It presented in detail the major problems we found while developing our method, justifying our approach. We discussed the necessity for both each intermediate structure used by ROOA and each main task within ROOA. During this discussion we presented the temporary solutions we adopted and gave the reasons which led us to reject them.

To summarise, this chapter finished by reviewing the three main versions of the ROOA method since the beginning of this project. The final version of ROOA is presented in Chapter 6.

Chapter 8

Assessment of the ROOA Method

8.1 Introduction

The ROOA method is the final product of our work while investigating the advantages of adding formality into the object-oriented analysis process. Existing OOA methods suffer from two fundamental problems: weak integration between the static, dynamic and functional models and lack of formality. We designed ROOA so that this gap could be bridged.

A rigorous method should satisfy a number of properties, namely proofs. These properties are difficult, if not impossible, to impose in an analysis method. In this chapter we analyse this problem and assess:

- the creation process as set out by the ROOA method;
- the understandability of the LOTOS specification produced by ROOA;
- the suitability of LOTOS as the formal specification language chosen.

Finally, we review the main findings that we made while observing ROOA being applied to several problems, by ourselves and by others.

8.2 Why a Rigorous Object-Oriented Analysis Method?

Given that, in order to be useful, software has to be correct, reliable and efficient, it is clear that we only gain by using formal techniques. Our work is concerned with enhancing the object-oriented analysis process. We know that the object-oriented analysis methods that are currently used suffer from several deficiencies, namely:

- they are weak at representing dynamic views;
- they do not integrate the static, dynamic and functional properties;
- they lack formality in the method and in the models used;
- they need supporting tools to check the semantics of the models.

The use of a formal specification language, such as LOTOS, during the analysis phase can overcome these deficiencies. As a matter of fact, LOTOS is good at describing the behaviour of a system (after all that is what it was designed for); the ROOA method teaches us how to write LOTOS specifications which integrate the static, dynamic and functional properties of a system; LOTOS has a precise syntax and semantics, and therefore the resulting ROOA model is formal; LOTOS has supporting tools, such as syntax and static semantic checkers and simulators, which can be used to check that the final model is semantically correct and to validate the specification against the requirements.

The validation of the specification against the requirements cannot be done by using formal proofs, because the original requirements are informal. In ROOA, however, we validate by executing the LOTOS specification, by using prototyping with sets of interface scenarios. Each scenario requires from the entire system a wide range of services. By executing the specification with the whole set of scenarios we expect to validate, at least, the more important requirements from the client.

8.3 What About Rigorous Methods?

It would be good to be able to reason formally about the requirements of a new software system as soon as possible. If this was possible, we could prove that a specification met the requirements and also use prototyping to refine the requirements and to correct errors, ambiguities and inconsistencies early. However, is it possible to develop an approach which formalises the analysis process?

The answer is 'no'! A formal approach tells us to develop programs from a given formal specification and how to verify these programs to meet the specification [BG77, BJ83, EM85]. When using formal approaches, the correctness of programs is established by means of mathematical proofs [Jon80]. If the input is a formal specification, we can propose a formal development process from there. For example, it is possible to provide a formal design process if a formal requirements specification already exists. However, the analysis process starts from a set of informal requirements and may include the capture of the requirements, involving discussions with clients or the users of the system. Thus, an analysis method cannot provide a formal process, as we cannot expect that our clients are all mathematicians, able to express their need by means of a set of equations, for example.

Nevertheless, we can reduce the distance between informal requirements and formal specifications by providing a process which creates formal specifications earlier than usual. Shortening this distance is the main goal of our work, and the primary result of the ROOA method. The final specification provided by ROOA is a requirements specification which can then be used as the starting point of a formal design process, as mentioned above.

The following are the characteristics that make ROOA a rigorous method:

- ROOA produces a formal requirements specification, expressed in the specification language LOTOS which is formal and has a clear mathematical semantics.
- ROOA uses prototyping to validate the LOTOS specification against the requirements.
- ROOA proposes algorithms to be followed when constructing the intermediate and final structures.

  - The object communication table can be produced automatically from the information in the object model.
  - The object communication diagram follows an algorithm to be produced from the object communication table, so it could be created automatically.
  - A skeleton of the LOTOS processes can be created automatically from the OCT.
  - The top-level behaviour expressions of the specification can be automatically generated from the object communication diagram by following an algorithm.

- ROOA provides a systematic development process, by offering a set of well-defined steps, heuristics and mappings from object-oriented concepts into LOTOS.
- ROOA builds on well-established methods and tools.

Therefore, ROOA is not a formal method. We call it a *rigorous method*, as it is less formal than a formal method should be. But, as America said during a panel at OOPSLA'91, formality is not a goal in itself, but is only useful as a means towards more efficient and more reliable software development [dCAC +91].

We think that we have achieved an important goal with our work: we developed a means of creating a formal requirements specification. This formal requirements specification can then be used as a starting point for a more formal development.

During this thesis, and following the terminology in the literature, we have been using the term *formal method* when referring to a specification languages such as LOTOS. A method includes a set of notations together with a strategy to be followed and heuristics. Formal methods as such do not exist yet [dCAC +91]. What does exist are what we can call formal notations, or formal specification languages, or formal description techniques, or formal techniques.

8.4 Strengths and Weaknesses of the ROOA Method

8.4.1 The ROOA Process

Bringing formal methods to the analysis phase of the development process is a new area of research and little work has been done so far. ROOA is our contribution.

The first major strength of ROOA is that it combines two important software development techniques: object-oriented analysis and formal methods. ROOA promotes formal methods in an area where they are not yet being used and, on the other hand, it adds formality to the informal object-oriented analysis methods. The specification resulting from an application of ROOA acts as an initial formal requirements specification. This specification can be used as the starting point of a formal development trajectory where a requirements specification is transformed into a design specification either by using correctness-preserving transformations or by using prototyping to ensure that the two specifications conform to one another [CJ92].

ROOA builds on the work already available for object-oriented analysis methods. By using an executable specification language such as LOTOS, ROOA produces a prototype where the LOTOS simulators may be used as the prototyping tools, which may be used for validating the requirements.

ROOA has some weaknesses too. Some of these weaknesses can be avoided by changing parts of the method. For instance, ROOA starts with the object model. This is not a weakness in itself, but favours systems with a strong static component. If the system has a strong dynamic component, we may find it difficult to produce an object model. Currently, we propose the use of event trace diagrams to help in capturing the message passing between structural components of the system and then use that information to build the object model. Notice that LOTOS is primarily a dynamic model and so that kind of problem only appears in the early stages of ROOA. To solve this problem we could modify Task 1 to emphasise an informal dynamic model to start with.

Some may consider the use of LOTOS to be a weakness of ROOA. We believe however that ROOA can easily be adapted to embrace other specification languages. However, more work is needed in this direction.

Finally, ROOA is not a formal method. We do not see how to create a purely *formal* process for this phase of software development. ROOA proposes a set of tasks to *systematically* produce an initial formal requirements specification. During the method we give rules to be followed, and we can build tools which can automate part of the ROOA process of building the object communication tables, the object communication diagrams and a first outline of the LOTOS specification.

8.4.2 Importance of Techniques Used within ROOA

Task 1: Build the Object Model. In order to build the object model we can use any of the existing OOA methods. Depending upon the type of the problem we can start identifying candidate objects in the requirements or, if the problem is strongly dynamic, we may begin by using techniques such as event trace diagrams.

Task 2: Refine the Object Model. Refining the object model means to guarantee that the object model produced in Task 1 has class templates defined with attributes and services, has interface objects, has relationships and message connections between objects. This is a major task in ROOA, stimulating important discussions about the behaviour of class templates and objects. During this task we build the OCT which is an important technique to bridge the gap between the refined object model and the OCD and the LOTOS specification.

Object Communication Table (OCT). The OCT collects the class templates, the services offered, the services required, the clients of each offered service, and the gates used for communication. Also, it helps us to reason about the necessary services each class template offers and, for each offered service, which other services are required. We find that one of the most helpful characteristics of the OCT is the help it provides in exploring the gates for communication between processes in the LOTOS specification.

Identifying offered services and required services is useful to help us establish the channels of communication between objects.

Object Communication Diagram (OCD). The OCD is a graph which shows objects communicating between each other by means of gates. It gives a good summary of the structure of the LOTOS specification. This makes the specification easier to understand, especially for lengthy specifications.

We give a set of rules to build the OCD from the OCT. From the OCD we can then use the algorithm developed by Clark [Cla94b] to create the high-level LOTOS behaviour expressions, showing all the gates and the processes interacting. The only case where this algorithm cannot be followed is when there are closed cycles of interacting objects and this does not occur often in practice.

The OCD plays a significant role in the ROOA method. In problems with many class templates, it is most useful to provide a "first-cut" between the informal model, represented by the object model and the OCTs, and the formal specification. On the other hand, it may not seem very useful if we are analysing a small problem with a few class templates.

Modelling the OOA Concepts. ROOA offers a mapping on how to model OOA concepts in LOTOS. The main concepts we deal with are: class templates, services, attributes, classes, abstract classes, objects, object identity, message connections, aggregates, subsystems and inheritance. LOTOS can model incremental inheritance and behavioural inheritance. Behavioural inheritance is described by Clark in [Cla94c] who then shows how it can be specified in LOTOS [MC93a, CM94].

8.4.3 Assessing the Resulting LOTOS Specification

While assessing the LOTOS specifications produced by ROOA, two main questions come to mind:

- Do LOTOS specifications reflect the analysis model being created?
- Are LOTOS specifications readable?

LOTOS has a very rich set of constructs which allow us to express many different ideas. For example, the parallel operators are good at showing the final specification as a set of concurrent objects, just the way we can imagine them in the object model. The distinction between pure synchronisation, synchronisation with value passing and interleaving are of great value when writing a specification. Non-determinism and value generation are also very useful concepts which allow us to abstract from detail and avoid design decisions at this stage.

Also, LOTOS achieves a clear separation of concerns between behaviour and data. Behaviour is described in the process part. This can be done in an elegant way that is not difficult to understand. Moreover, the process part of the specification preserves the structure of the static object model, and this is important for improving the understandability of the final result. On the other hand, data is modelled in the abstract data type part. As we said before, defining abstract data types can be a laborious and time consuming task. However, the simplifications we have proposed in Chapter 3 make this task easier and faster. It is in the abstract data type part that the internal structure and local state of objects are fully defined. This state is hidden from the other objects. The only means of communication between objects is by message passing and this happens in the process part.

The difficulties of understanding formal specifications are a problem shared by most, if not all, formal specifications languages. One could say that it is probably easier to specify a problem than to read and understand the specification created by someone else. But, is

this not also the case for programming languages? Yes, it is! So, in which way are formal specifications more difficult to understand than programs? We believe that a specification language is not necessarily easier or harder to master than a programming language. A program in C, a program in Ada and a program in Prolog look very different from each other. Each of these programming languages requires special skills from the programmer. Why should a formal specification language be different?

We believe that the problem in writing specifications is getting the right level of abstraction. When writing programs we use a low level of abstraction, as compared with what is usual in analysis, and we spend much time in so-called "implementation details". When writing specifications we use a higher level of abstraction where the details we have to consider are of a different kind. Hall observes that many people find it difficult to write specifications because it is difficult for them to get away from the detailed descriptions they are used to when writing programs [Hal90].

In order to make our LOTOS specifications easier to read and to understand, we propose an object-oriented style together with meaningful names for processes, events, abstract data types, operations, parameters and gate names, supplemented with informal comments before the specification of each process and ADT. Meaningful names do not always come to mind immediately; however, we cannot overlook this point, as requirements specifications can be used for communication between users, analysts and designers. Moreover, the ROOA-style maintains a correspondence between the concepts and names used in the object model, the OCTs and OCDs, and the final LOTOS specification.

8.5 Suitability of LOTOS

In Chapter 2 we discussed the reasons that led us to choose LOTOS as our formal description technique used by ROOA. The characteristics of LOTOS which make it the obvious candidate are:

- LOTOS is an ISO standard;
- LOTOS is able to produce specifications in an object-oriented style;
- LOTOS support tools are available;
- LOTOS specifications are executable and so prototyping can be used;
- LOTOS supports concurrency.

However, LOTOS is not ideal for this task. It has shortcomings which we will discuss now. The first criticism is that LOTOS does not directly support object-oriented constructs: neither simple ones, such as class templates, nor more complex ones, such as inheritance or aggregates.

Another criticism is that specifying abstract data types in LOTOS is a long and tedious task, even if it is not too difficult. All algebraic specifications seem to suffer from the same problem, but it could be alleviated if LOTOS was to offer more built-in facilities. Abstract data types rapidly increase the size of a LOTOS specification. Notice however that the solution we propose for specifying abstract data types cuts much of the work we have to do. A related problem is that LOTOS is not particularly good at handling calculations. As

a matter of fact, LOTOS has a very simplistic approach to arithmetic. Even naturals have to be explicitly defined using the basic constructors 0 and *succ*. Also, LOTOS does not incorporate an input/output interface. We cannot read values or write values. Everything is done by value passing. This is one of the reasons for defining interface scenarios, because, as we explained before, an interface scenario contains calls to the system and receives the respective answers.

The size of algebraic specifications, and therefore also LOTOS specifications, tends to run into dozens of pages even for simple to average problems. It is difficult to ensure by hand that specifications this large are correct and consistent, especially when an incremental development is being applied, such as the one taken by ROOA. This is a point where the LOTOS tools are most valuable. The syntax checkers, static semantic checkers and simulators help us to identify several kinds of errors in a specification, including deadlock. The capability to execute LOTOS specifications is definitely a great advantage, since it helps us to gain confidence in our model.

Formal specifications are difficult to understand at first glance for a newcomer (the same can be said for programs, of course). This is not only a LOTOS problem, but in LOTOS we can certainly improve the readability of the final formal specification by writing LOTOS in a 'ROOA-style', as mentioned in Section 8.4.3.

A general comment we have received from reviewers of our papers is that LOTOS is too low-level a language; it is too close to a programming language. We do not think that this is a bad thing in itself. As far as we know, all specification languages which produce executable specifications share this feature.

Even with its shortcomings, we still think that LOTOS is a good choice to be used within the ROOA method for many kinds of problems, and we believe this thesis has demonstrated this view.

8.6 ROOA Applied to Case Studies

While developing a new analysis method, it is wise to apply it to many different kinds of problems. Otherwise, we cannot be sure that different types of requirements can be dealt with appropriately within the new approach. For ROOA, it was important not only to apply the method to a variety of systems, but also to use different object-oriented analysis methods within it. On the one hand, by applying ROOA to different problems we could identify aspects of ROOA which should be improved, in order to support certain characteristics of systems. On the other hand, by applying ROOA together with different OOA methods to the same problem and to different problems we could identify better the distinguished features of the methods which have to be taken into account. This makes ROOA compatible with several kinds of existing object-oriented analysis methods.

ROOA has been applied to database oriented problems with simple communication patterns and to problems with a more complex dynamic behaviour. Also, it was used together with the methods by Coad and Yourdon [CY91a], Rumbaugh *et al.* [RBP+91] and Jacobson [Jac92]. This helped us identify system development requirements and object models with which ROOA had be compatible.

In this section we discuss briefly the applications of the ROOA method to problems. This includes both our own experiences and other's people experiences while using ROOA. The results of those experiments and and the lessons learnt through them were incorpo-

rated into ROOA as it evolved.

A full description of case studies is given in the technical report *Specification Case Studies in ROOA* [MC94e]. As with all new methods, it is only possible for the reader to fully understand the method when he or she can see it applied to problems. Here, we are only giving a brief discussion; the reader is strongly recommended to study the technical report.

8.6.1 The Automated Banking System

In this thesis we have been using as an example and a guide, the specification of an automated banking system. We analysed the problem using a number of different methods and with different requirements. First we used a sequential version of the problem. Only then did we extend the requirements to include more objects and to support concurrency.

The exercise helped us to develop the major features of ROOA, namely, how to specify the object-oriented concepts in LOTOS. During the analysis we found many simple, common objects, such as accounts, which can take advantage of inheritance and be specified as a process definition with some guards. It also had other more complex objects which were not so straightforward, such as cheques. However, the banking system has a simple pattern of communication; therefore we had to be careful not to overadapt ROOA to problems with a strong static component.

For the initial analysis we first used the method of Coad and Yourdon. Later we redid the problem using the method by Rumbaugh *et al.* By using different OOA methods, even with the same problem, we identified the need to improve ROOA by adding Task 2 (Refine the Object Model). This is because the existing OOA methods differ in the amount of information they show in the object model. For instance, while object models produced by using Coad and Yourdon's method show class templates with attributes and services, conceptual relationships and message connections, the method by Rumbaugh *et al.* does not show services and messages connections. Other object models differ even more. This is the case with object models produced by Jacobson's method, for instance, which only show class template names with relationships and message connections.

It was while we were analysing the banking system that we divided Task 3 (Build the LOTOS Formal Model) into its major subtasks. The composition of the subtask *Refine the Specification* suffered significant changes during this time. But, once established, there were no major changes proposed.

8.6.2 The Warehouse Management System

After we finished the automated banking system problem we realised that we needed a new problem with the following characteristics:

1. The new problem should have a stronger dynamic component than the banking system.

2. The object model from Task 1 should look different.

3. The object model might have been created by someone else.

For these reasons, we took the warehouse management system described by Jacobson [Jac92]. We changed the object model presented in Jacobson's book since it already took

into consideration a system of window interfaces. We eliminated some of the class templates, for example `Warehouse Truck Radio` and `Truck Radio`, as they were not of much use for our exercise. The warehouse problem has a more complex pattern of communication than the banking system. This led us to use auxiliary techniques that could help us to think earlier about communication between objects. This way we were able to start using LOTOS sooner, LOTOS being a good language for specifying the behaviour of systems.

Jacobson's object model only shows the name of the class templates, static relationships and message connections. This is directly related to points 2 and 3 referred above. On the one hand, the object model looks different from an object model produced by using the method by Coad and Yourdon or the one by Rumbaugh *et al.* On the other hand, the object model was not produced by ourselves.

We found many difficulties in understanding Jacobson's object model. In our opinion a class template cannot be fully understood until we define its services and attributes (although Jacobson does not include them). We used the use cases to identify the information that was missing in the object model, but this was a difficult task. The warehouse management system helped us to strengthen Task 2, in particular the subtasks which are concerned with identifying services and attributes, and with building the OCT. The OCT proved itself to be a valuable technique for collecting all the information about services and communications between objects. This was also an opportunity to validate the algorithm used for building the OCD.

In retrospect, the warehouse problem helped us to propose a more robust Task 2, capable of accepting a wider range of object models, showed the valuable use of OCTs and OCDs, and highlighted the adequacy of LOTOS to specify concurrency and communication between objects.

8.6.3 The Car Rental System

The two systems described in the two previous sections helped us to develop and to improve the ROOA method. Having ROOA in its final stage, we felt it would be good to apply it to a new problem, so that we could verify whether some other small changes were necessary.

The new problem was an automated car rental and billing system for a car rental company. Briefly, this company has several branches, each one with several terminals connected to the main software system. Each branch has cars associated. Cars can be reserved (and cancelled), rented and then returned to the same branch where they were rented or to any other branch of the company.

Applying ROOA to this problem was a useful exercise, to make us feel more comfortable with our method. We tested how the various ROOA techniques interact and help each other as validation tools. In particular, we tested each task and subtask. The car rental system has some interesting communication patterns. This helped us to evaluate once more the need for the event trace diagrams, as an intermediate technique to help building the object model. The event trace diagrams were also very useful in helping us build the object communication tables which then were the major source of information in the creation of the object communication diagram.

There is an important conclusion from this application of the ROOA method: object model, event trace diagrams, object communication tables, object communication diagram and the LOTOS specification were being used many times to improve each other. All these techniques are integrated in the sense that, at a given stage, we were using all of

them at the same time, in a parallel and interactive way. For example, from the event trace diagrams and the object-communication table we started writing LOTOS code. As coding advanced, we identified more services and communications which were not yet fully identified in the previous tasks.

8.6.4 ROOA Applied by Others

The ROOA method has also been applied by other people. It was gratifying to see that the method can be understood and applied effectively by other people.

Clark applied ROOA to a second version of the warehouse problem [Cla93]. As a result of his work, we introduced event trace diagrams in the method. The method of Rumbaugh *et al.* uses event trace diagrams to build the dynamic model. We had decided to give preference to LOTOS as the technique to describe behaviour of objects and communication between them. Clark's results showed that the use of a simple technique such as event trace diagrams could be of great help to understand the communication between objects, without having to deal too soon with the intricacies of a specification language.

We had a M.Sc. student assessing ROOA for three months. In her dissertation, Lim [Lim93] applied our method to two problems: a simple restaurant problem and a hotel reservation and billing system. The first case study was composed of six class templates, without complex structures such as inheritance and aggregation. Its goal was to help Lim learning about OOA methods and LOTOS, but little about ROOA. Because it was a small problem, it was not possible to transmit to the student the necessity of the OCDs, for example.

The second case study was composed of fourteen class templates, five of them related by an aggregation relationship. This problem was used to undertake a more thorough and critical assessment of the ROOA method and of its intermediate structures. It was from the discussions with Lim about this case study that the necessity for an earlier version of the OCTs was identified. Here is an extract of Lim's evaluation of ROOA [Lim93]:

> The use of LOTOS [in the ROOA method] brings with it the advantage of language support tools and the ability to execute the specifications to simulate the requirements specified.
>
> Standard LOTOS is a good fit for specifying an object-oriented method and work has been done [MC93c] to show how OOA concepts in the object model can be specified using LOTOS. However, like all rigorous techniques, it brings with it the need to have a special training for using and understanding it. [...] The guidelines and templates in ROOA facilitated the learning of LOTOS and its use in the modelling of the case study requirements. Examples such as the Bank Account example [MC93d], were found to be especially helpful as they provided a complete picture of how the templates fitted in the entire specification.
>
> Having a rigorous technique for specifying requirements in object-oriented technology will enable highly mission-critical software to make use of object-oriented methods. [...]

Having Lim with us applying ROOA was very important, as she was at the same time learning about OOA methods, LOTOS and ROOA. This gave us a good feel of how

a newcomer would face a method such as ROOA which, while not using mathematics directly, used a formal specification language. Lim's work together with the questions we had to answer during these three months have been of great help in improving both ROOA and the readability of the available document describing the method.

Lim did learn LOTOS, object-oriented analysis methods and ROOA (and wrote her dissertation). This is a sign that, after all, formal specification languages, LOTOS at least, are not so difficult to learn and that the ROOA method can be understood and used by others besides ourselves.

8.7 ROOA: Domains of Application

ROOA can be used to model the same kinds of systems modelled by other OOA methods. Since it uses LOTOS, ROOA is also suitable to model the kinds of systems usually modelled in LOTOS. Therefore, we can for example use ROOA to model concurrent, database management systems, and network communication systems.

8.8 Conclusions

This chapter together with Chapter 7 presented the main ideas behind ROOA. While Chapter 7 presented the evolution of the ROOA method, justifying the current version, this chapter began by discussing the properties that rigorous object-oriented analysis methods should display, then it evaluated ROOA, by discussing its weaknesses and strengths and by assessing the resulting LOTOS specification.

Finally, this chapter also examined the suitability of LOTOS as the chosen specification language and discussed both our experiences while applying ROOA to case studies and the results obtained by other people while testing ROOA.

Chapter 9

Conclusions and Prospects

This chapter reviews the goals of this thesis, discusses how those goals were pursued and achieved by our work, and suggests possible future developments.

9.1 Summarising the Goals of the Thesis

The primary objective of our work was to investigate how formal methods can be used in combination with object-oriented analysis methods during the analysis phase of the software life cycle. To meet this objective, we addressed five separate goals:

- assessing the existing OOA methods to identify their differences and similarities;
- examining the LOTOS specification language and finding a way to use LOTOS in an object-oriented style;
- devising the possibility of creating a formal requirements specification during the analysis phase;
- combining LOTOS with OOA methods, by means of a rigorous analysis method;
- interpreting the object-oriented concepts formally, independently of the LOTOS language.

9.2 Results of the Thesis

Chapter 2 introduced object-oriented analysis methods and formal specifications. It identified the general features of OOA methods and motivated the early use of formal methods so that a formal requirements specification can be created sooner than usual. This chapter defined the goals that a formal method should aim for so that a formal object-oriented specification can be produced, and explored the idea of prototyping, discussing how it can be used within the ROOA method.

Chapters 3 and 4 described how to model standard object-oriented analysis concepts in LOTOS. This includes the representation of objects as LOTOS processes with abstract data types. The LOTOS process specifies the dynamic behaviour of the object, and the abstract data types, given as parameters of the process, specify its state information. If

the object merely plays the role of an attribute of another object, it is specified as a single abstract data type. An abstract data type contains only the equations necessary to allow the objects to be prototyped with state information and values to be exchanged during communication.

Two different definitions of a class occur in the literature. We distinguish between: (a) a *class template*, used to represent common features of objects of the same kind; and (b) a *class*, used to represent a collection of objects. In LOTOS a class template is represented by a process definition, and a class is represented as an object generator. An object is a member of a class and is created by instantiating a class template. It is referenced by using an object identifier. Object identifiers are instances of a special LOTOS abstract data type, which we have added to the LOTOS library. Each time an object is created, we use value generation to "produce" a new identifier for that object.

Interactions between objects (message connections) are represented by LOTOS process communication constructs. The primitive LOTOS communication construct is *event synchronisation*, in which two processes synchronize on a *gate*, and may exchange data. Complex object interactions may be built out of simpler interactions, by using the LOTOS composition operators.

To define inheritance in the LOTOS framework we use technical features of LOTOS, namely abstract superclasses with exit functionality. We model conceptual relationships by means of attributes, which can be an object identifier or a set of object identifiers. The associations with values are modelled as new objects.

Chapter 4 was concerned with aggregates. It reviewed the definitions of aggregates from other authors and presented our own view of aggregates and how we model them in LOTOS by using the LOTOS parallel composition operators. We consider aggregation as an important concept to scale up a system. This is the reason why we dedicated a whole chapter to study that concept.

Chapter 5 was devoted to the formal definition of the basic OOA concepts in terms of simple mathematical formalisms, such as tuples and functions. The major contribution of this chapter is to show how the behaviour of objects belonging to the running system can be derived from the generic definition of the behaviour of the class template. We do that in a new way, which is different from the approaches followed by most authors, when trying to give a formal interpretation of an object-oriented specification.

Chapter 6 presented the main result of our work: the Rigorous Object-Oriented Analysis (ROOA) method. ROOA enables a formal object-oriented analysis model to be devised from a set of informal requirements, and results in a formal requirements specification expressed in LOTOS. ROOA consists of three main tasks: building an object model, refining the object model, and building the formal LOTOS OOA model. Each of these tasks involves multiple passes through subtasks. The three tasks are not necessarily sequential: some parts of the model may be built through to the LOTOS specification before other parts of the model are analysed.

The first task, building the object model, may be accomplished in the first pass by using any of the usual object-oriented analysis methods, such as the methods of Coad and Yourdon [CY91a], Rumbaugh *et al.* [RBP $^{+}$91] and Jacobson [Jac92]. The object model is refined in the second task by passing through three subtasks: completing the object model, identifying the initial identification of dynamic behaviour, structuring the object model. The formal LOTOS OOA model integrates the static, dynamic and functional properties of the system, and consists of five subtasks: creating the object communication

diagram (OCD); specifying the class templates as LOTOS processes and ADTs; composing objects; prototyping the object model by executing the LOTOS specification; and refining the specification according to the results of this rapid prototyping.

As LOTOS has a precise mathematical semantics, the resulting model is formal and unambiguous. Moreover, as LOTOS is executable, the model is executable, and so prototyping can be used to give immediate feedback to clients.

We believe that Chapter 6, together with Chapters 3, 4 and 5 constitute the main contributions of this work.

Chapter 7 described the evolution of the ROOA method during its development. It explored the problems we faced, the temporary solutions we adopted and the reasons for moving beyond them. This is useful to appreciate the reasons why ROOA is the way it is.

Chapter 8 presented a critical assessment of the ROOA method and of LOTOS as the specification language used. It addresses the question of rigorous methods versus formal methods. Together with Chapter 7, this chapter gives a basis for understanding the main ideas behind the design of ROOA.

9.3 Future Work on ROOA

9.3.1 Improvements in the ROOA Method

The development of ROOA is not complete. One area of further investigation is concerned with Task 1, which could be modified to support the initial analysis of a system with a strong dynamic component. We also need to devise more specific rules for grouping objects into subsystems and develop the concept of communication between objects of the same class.

A related area of work would be to modify Task 3 (Build the LOTOS Formal Model) so that other specification languages can be used. This would involve redoing the modelling of object-oriented concepts, presented in Chapter 3 and Chapter 4, in the new language.

Another area of further research is related to reusability. The specification of conceptual relationships as parameters of the class templates involved in the relationship seems to go against the idea of reusable objects. There are situations in which two objects exist in the real world with a specific relationship, but in general, an object exists independently of its relationships. In this document, the modelling of relationships blurs the definition of class template, making it more difficult to be used as a reusable component. This problem needs to be dealt with, and a strategy to create a library of reusable components has to be investigated.

In the last couple of years researchers have been more and more interested in *patterns* as a technique leading to reusability. Patterns are groups of repetitive objects and relationships between these objects, which are likely to occur many times in the same or in different applications. We are interested in studying how ROOA can handle them.

Finally, we have not yet investigated the optimisation of the various techniques within ROOA. For example, we do not yet have an algorithm for minimising the number of gates in the object communication diagram. These issues are subjects for further research work.

9.3.2 Useful Tools to Support ROOA

We propose the development of several tools which can support the application of the ROOA method.

An OCT generator would take the textual information in an object model and create the object communication tables. Usually, the object model only contains the class templates with attributes and services and relationships and message connections between objects. However, tools exist that allow us to record information in the object model about class template, services, attributes, etc. Therefore, instead of building the OCT by hand, a translator could be used. An initial version of such a translator has been developed by Clark [Cla94a].

An OCD graphical tool would take the information in the OCT and produce an object communication diagram. This tool should allow entities (such as class template and arcs between them) in the diagram to be moved, maintaining the connections of the entity moved with the rest of the entities in the diagram. A prototype version of this tool is being developed during the summer of 1994 by Chua [Chu94].

A LOTOS generator would take the information in an OCT and produce a first outline of the LOTOS specification for each class template. Recall that the names of the processes are in the first column, the gates in the fifth column and the services in the second. The services can be offered in the corresponding LOTOS process as structured events, in choices of the '`[]`' operator.

A top-level behaviour expression generator to create the LOTOS top-level behaviour expression from an OCD.

A CASE tool could be developed to integrate the LOTOS tools with the tools described above and to give us a common base of information.

9.3.3 Broader Applications

During the three years of our research we had to divide the time between research, writing papers and technical reports, improving the ROOA method while applying it to case studies. We would like to have spent more time applying ROOA to more problems, but certainly that would have decreased the time available to write and publish technical papers and reports.

For the future, we plan to apply ROOA to different types of problems, such as transaction-based systems and real-time systems. We would also like to bring together a team that would try to apply ROOA to a pilot project for industry. This would help to identify areas in which the method is weak and needs further improvement.

9.4 Concluding Remarks

In summary, this thesis has shown that it is possible to combine formal specification languages with existing object-oriented analysis methods in a practical and effective way.

We believe that it is possible to apply ROOA to create an initial formal requirements specification which can then be used as the starting point of a formal development strategy.

Bibliography

[AJ90] H. Alexander and V. Jones. *Software Design and Prototyping Using me too.* Prentice-Hall, 1990.

[Ame91] P. America. Designing an Object-Oriented Programming Language with Behavioural Subtyping. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, 1991.

[AP93] S. Austin and G.I. Parkin. Benefits, Limitations and Barriers to Formal Methods. Technical report, Division of Information Technology and Computing, National Physical Laboratory (NPL), United Kingdom, March 1993.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[BB89] B.W. Boehm and F.C. Belz. Applying Programming to the Spiral Model. In C. Tulley, editor, *Proceedings of the 4th International Software Process Workshop: ACM Software Engineering Notes*, 14(4):46-56, June 1989.

[BCG83] R.M. Balzer, T.E. Cheatham, and C.C. Green. Software Technology in the 90's: Using a New Paradigm. *IEEE Computer*, 16(11):39–45, November 1983.

[BD92] T. Bar-David. Practical Consequences of Formal Definitions of Inheritance. *Journal of Object-Oriented Programming*, 5(4):43–49, July/August 1992.

[BDMN80] G.M. Birtwistle, O.J. Dahl, B. Myrhaug, and K. Nygaard. Simula Begin. Lund, Sweden: Studentlitteratur, 1980.

[Bel86] F.C. Belz. Applying the Spiral Model: Observations on Developing System Software in Ada. In *1986 Annual Conference on Ada Technology*, pages 57–66, Atlanta, Georgia, 1986.

[Ben56] H.D. Benington. Production of Large Computer Programs. In *ONR Symposium on Advanced Program Methods for Digital Computers*, pages 15–27, June 1956. Also in Annals of the History of Computing, October 1983, pp350-361.

[Ben83] H.D. Benington. Production of Large Computer Programs. In *Annals of the History of Computing*, pages 350–361, 5(4), October 1983.

[Ben88] J. Bentley. Bumper-Sticker Computer Science. In *More Programming Pearls*. Addison-Wesley, 1988.

[Ber88] E. Berard. Object Oriented Development for Ada Software. EVB Software Engineering Inc., 1988.

[Ber89] E. Berard. Object Oriented Requirements Analysis. EVB Software Engineering Inc., 1989.

[Ber93] E.V. Berard. *Essays on Object-Oriented Software Engineering*, volume I. Prentice-Hall, 1993.

[BG77] R. Burstall and J.A. Goguen. Putting Theories Together to Make Specifications. In *5th IJCAI*, pages 1045–1058, Cambridge, Mass, 1977.

[BGHS91] G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd. *Object-Oriented Languages, Systems and Applications*. Pitman, 1991.

[BGS84] B.W. Boehm, T.L. Gray, and T. Seewaldt. Prototyping Versus Specifying: A Multiproject Experiment. *IEEE Transactions on Software Engineering*, SE-10(3):290–302, May 1984.

[BGW82] R.M. Balzer, N.M. Goldman, and D.S. Wile. Operational Specification as the Basis for Rapid Prototyping. *ACM Software Engineering Notes*, 7(5):3–16, December 1982.

[Bha83] K.S. Bhaskar. How Object-Oriented is your System. *SIGPLAN Notices*, 18(10):8–11, October 1983.

[BJ83] D. Bjørner and C.B. Jones. *Formal Specification & Software Development*. Prentice-Hall, 1983.

[Bla89] S. Black. Objects and LOTOS. Technical report, Hewlett-Packard Laboratories, Stoke Gifford, Bristol, 1989.

[Bla93] M. Blaha. Aggregation of Parts of Parts of Parts. *Journal of Object-Oriented Programming*, 6(5):14–20, September 1993.

[BM85] G.D. Buzzard and T.N. Mudge. Object-Based Computing and the Ada Programming Language. *IEEE Computer*, 18(3):12–19, March 1985.

[BMS93] J.P. Bahsoun, S. Merz, and C. Servieres. A framework for programming and formalizing concurrent objects. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering: ACM Software Engineering Notes*, 18(5):126-137. ACM Press, December 1993.

[Boe81] B.W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[Boe87] B.W. Boehm. Industrial Software Metrics Top 10 List. *IEEE Software*, 4(5):84–85, September 1987.

[Boe88] B.W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[Boo87] G. Booch. *Software Engineering with Ada*. Benjamin/Cummings, 2nd ed. edition, 1987.

[Boo91] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, 1991.

[Bri94] T.L. Briggs. A Specification Language for Object-Oriented Analysis and Design. In M. Tokoro and R. Pareschi, editors, *ECOOP'94*, volume 821 of *Lecture Notes in Computer Science*, pages 365–385. Springer-Verlag, 1994.

[Bro75] F.P. Brooks. *The Mythical Man-Month — Essays on Software Engineering*. Addison-Wesley, 1975.

[Bro91] D. Brookman. SA/SD vs OOD. *Ada Letters*, 11(9):96–99, November/December 1991.

[BT75] V. Basili and A. Turner. Iterative Enhancement: A Practical Technique for Software Development. *IEEE Transactions on Software Engineering*, 1(4):390–396, December 1975.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremeas. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.

[CDD+89] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An Object-Oriented Extension to Z. In S.T. Vuong, editor, *Formal Description Techniques, II*, pages 281–295. North-Holland, 1989.

[CGR93] D. Craigen, S. Gerhart, and T. Ralston. An International Survey of Industrial Applications of Formal Methods — Purpose, Approach, Analysis, and Conclusions. Technical Report NISTGCR 93/626, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, March 1993.

[Che76] P. Chen. The Entity Relationship Model: Toward a Unifying View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[Cho92] C. Choppy. Prototyping and Formal Specifications. In C.M.I. Rattray and R.G. Clark, editors, *The Unified Computation Laboratory — Modelling, Specifications, and Tools*, pages 141–170. Oxford University Press, 1992.

[Chu94] K.C. Chua. Interviews and Object-Oriented Design. Master's thesis, Department of Computing Science and Mathematics, University of Stirling, 1994. In development.

[Civ93] F. Civello. Roles for Composite Objects in Object-Oriented Analysis and Design. In *Proceedings of OOPSLA'93: ACM SIGPLAN Notices*, 28(10):376-393, October 1993.

[CJ92] R.G. Clark and V.M. Jones. Use of LOTOS in the Formal Development of an OSI Protocol. *Computer Communications*, 15(2):86–92, March 1992.

[CL91] E. Cusack and M. Lai. Object-Oriented Specification in LOTOS and Z or, my Cat Really is Object-Oriented! In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 179–202. Springer-Verlag, 1991.

[Cla90] R.G. Clark. The Design and Development of Embedded Ada Systems. *Software Engineering Journal*, 5(3):175–184, May 1990.

[Cla91] R.G. Clark. The Development of Concurrent Ada Systems from LOTOS Specifications. In R.J Mitchell and D. Simpson, editors, *Ada into the 90's*, pages 115–129. Woodhead Publishing Ltd, 1991.

[Cla92a] R.G. Clark. LOTOS Design-Oriented Specification in the Object-Based Style. Technical Report CSM-84, Department of Computing Science and Mathematics, University of Stirling, FK9 4LA Stirling, Scotland, April 1992.

[Cla92b] R.G. Clark. Using LOTOS in the Object-Based Development of Embedded Systems. In C.M.I. Rattray and R.G. Clark, editors, *The Unified Computation Laboratory — Modelling, Specifications, and Tools*, pages 307–319. Oxford University Press, 1992.

[Cla93] R.G. Clark. The Warehouse Problem: Personal Communication, 1993.

[Cla94a] R.G. Clark. An OCT Generator, 1994. Draft.

[Cla94b] R.G. Clark. Construction of LOTOS Behaviour Expressions from Network Diagrams. Technical Report CSM-124, Department of Computing Science and Mathematics, University of Stirling, Scotland, 1994.

[Cla94c] R.G. Clark. Inheritance and Reliability. In *Proceedings of TaTTOO'94: Teaching and Training in the Technology Of Objects*, Leicester, January 1994.

[CM94] R.G. Clark and A.M.D. Moreira. Behavioural Inheritance in ROOA. In R. Wieringa and R. Feenstra, editors, *Workshop on Information Systems - Correctness and reusability (IS-CORE'94)*, pages 346–356, Amsterdam, The Netherlands, September 1994.

[Col89] E. Colbert. The Object-Oriented Software Development Method: A Practical Approach to Object-Oriented Development. In *Proceedings of Tri-Ada'89 — Ada Technology in Context: Application, Development, and Deployment*, pages 400–415, 23(6), October 1989.

[Con89a] L. Constantine. Beyond the Madness of Methods: Systems Structure Modeling and Convergent Design. In *Software Development '89: Proceedings*. Miller-Freeman Publishing Co., 1989.

[Con89b] L. Constantine. Object-Oriented and Structured Methods: Toward Integration. *American Programmer*, 2(7/8):34–40, August 1989.

[CRS90] E. Cusack, S. Rudkin, and C. Smith. An Object-Oriented Interpretation of LOTOS. In S.T. Vuong, editor, *Formal Description Techniques, II*, pages 211–226. North-Holland, 1990.

[CY91a] P. Coad and E. Yourdon. *Object Oriented Analysis*. Yourdon Press, Prentice-Hall, 2nd edition, 1991.

[CY91b] P. Coad and E. Yourdon. *Object-Oriented Design*. Yourdon Press, Prentice-Hall, 1991.

[Dav82] A.M. Davis. Rapid Prototyping Using Executable Requirements Specifications. *ACM Software Engineering Notes*, 7(5):39–44, December 1982.

[Dav88] A.M. Davis. A Taxonomy of the Early Stages of the Software Development Life Cycle. *The Journal of Systems and Software*, 8(4):297–311, September 1988.

[Dav92] A.M. Davis. Operational Prototyping: a New Development Approach. *IEEE Software*, 9(5):70–78, September 1992.

[dCAC+91] D. de Champeaux, P. America, D. Coleman, R. Duke, D. Lea, and G. Leavens. Formal Techniques for OO Development. In *Proceedings of OOPSLA'91: ACM SIGPLAN Notices*, 26(11):166-170, November 1991.

[dCLF93] D. de Champeaux, D. Lea, and P. Faur. *Object-Oriented System Development*. Addison-Wesley, 1993.

[dCO89] D. de Champeaux and W. Olthoff. Towards an Object-Oriented Analysis Technique. In *7th Annual Pacific North West Software Quality Conference*, pages 323–338, 1989.

[DDBP93] E. Dubois, P. Du Bois, and M. Petit. O-O Requirements Analysis: An Agent Perspective. In O.M. Nierstrasz, editor, *ECOOP'93*, volume 707 of *Lecture Notes in Computer Science*, pages 458–481. Springer-Verlag, 1993.

[DeM79] T. DeMarco. *Structured Analysis and System Specification*. Prentice-Hall, 1979.

[DN66] O.J. Dahl and K. Nygaard. Simula — An Algol-based Simulation Language. *Communications of the ACM*, 9(9):671–678, September 1966.

[DT88] S. Danforth and C. Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[EGS91] H.-D. Ehrich, J.A. Goguen, and A. Sernadas. A Categorial Theory of Objects as Observed Processes. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 203–228. Springer-Verlag, 1991.

[EGS93] H.-D. Ehrich, M. Gogolla, and A. Sernadas. Objects and Their Specification. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, volume 655 of *Lecture Notes in Computer Science*, pages 40–65. Springer-Verlag, 1993.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications*, volume I. Springer-Verlag, 1985.

[ESS89] H.-D. Ehrich, A. Sernadas, and C. Sernadas. Objects, Object Types, and Object Identification. In H. Ehrig, H. Herrlich, H.-J. Kreowski, and G. Preuß, editors, *Categorical Methods in Computer Science*, volume 393 of *Lecture Notes in Computer Science*, pages 142–156. Springer-Verlag, 1989.

[EW93] H. Eertink and D. Wolz. Symbolic Execution of LOTOS Specifications. In M. Diaz and R. Groz, editors, *Formal Description Techniques, V*, pages 295–310. North-Holland, 1993.

[Fai85] R.E. Fairley. *Software Engineering Concepts*. McGraw-Hill, 1985.

[Fir91] D.G. Firesmith. Structured Analysis and Object-Oriented Design are not Compatible. *Ada Letters*, 11(9):56–66, November/December 1991.

[FK92] R.G. Fichman and C.F. Kemerer. Object-Oriented and Conventional Analysis and Design Methodologies. *IEEE Software*, 25(10):22–39, October 1992.

[Flo84] C. Floyd. A Systematic Look at Prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven, editors, *Approaches to Prototyping*, pages 1–18. Springer-Verlag, 1984.

[FMG89] M.M. Freitas, A.M.D. Moreira, and P. Guerreiro. Choosing Ada Development Methodologies for a High Reliability Message Switching System. In *4th AFCEA Hawaii Defense Electronics*, Hawaii, December 1989.

[FMG90a] M.M. Freitas, A.M.D. Moreira, and P. Guerreiro. Introducing Object-Oriented Methodologies with Ada in Portugal. In *International Conference of Asia Pacific Defense'90*, Seoul, Korea, November 1990.

[FMG90b] M.M. Freitas, A.M.D. Moreira, and P. Guerreiro. Object-Oriented Requirements Analysis in an Ada Project. *Ada Letters*, 10(6):97–109, July/August 1990.

[Fuc92] N.E. Fuchs. Specifications are (Preferably) Executable. *Software Engineering Journal*, 7(5):323–334, September 1992.

[Gen92] SPECS-Specification Generation. Final Methods and Tools for the Generation of Specifications. Technical Report 46/SPE/WP3/DS/A/008/b1, INESC, Lisbon, Portugal, December 1992.

[GFM89] P. Guerreiro, M.M. Freitas, and A.M.D. Moreira. Using Object-Oriented Design with Ada for a High Reliability Message Switching System. In *AFCEA International's Asia Pacific Defense'89*, pages 138–144, Seoul, Korea, September 1989.

[GHH+92] C. George, P. Haff, K. Havelund, A.E. Haxthausen, R. Milne, C.B. Nielsen, S. Prehn, and K.R. Wagner. *The RAISE Specification Language*. Prentice-Hall, 1992.

[Gib90] E. Gibson. Object — Born and Bred. *Byte*, 15(10):245–254, October 1990.

[GK76] A. Goldberg and A. Kay. Smalltalk-72 Instructional Manual. Technical Report SSL-76-6, Xerox Parc, Palo Alto, California, March 1976.

[GKK+88] J.A. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, and J. Meseguer. An Introduction to OBJ3. In J.P Jouannaud and S. Kaplan, editors, *Proc. Conf. on Conditional Term Rewriting*, number 308 in Lecture Notes in Computer Science, pages 258–263. Springer-Verlag, 1988.

[Gom83] H. Gomaa. The Impact of Rapid Prototyping on Specifying User Requirements. *ACM Software Engineering Notes*, 8(2):17–28, April 1983.

[Got93] R. Gotzhein. *Open Distributed Systems.* Vieweg Verlag, Wiesbaden, 1993.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and its Implementation.* Addison-Wesley, 1983.

[Gra89] D.R. Graham. Incremental Development: Review of Nonmonolithic Life-Cycle Development Models. *Information and Software Technology*, 31(1):7–20, January/February 1989.

[Gre89] C.C. Green. Personal Communication with P. Ladkin, 1989.

[GS79] C. Gane and T. Sarson. *Structured Systems Analysis: Tools and Techniques.* Prentice-Hall, 1979.

[GS81] H. Gomaa and P. Scott. Prototyping as a Tool in the Specification of User Requirements. In *5th International Conference on Software Engineering*, IEEE Computer Society Press, pages 333–342, Washington D.C., 1981.

[Hal90] A. Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, September 1990.

[Hed93] M. Hedlund. The Integration of LOTOS with an Object Oriented Development Method. In J.C.P. Woodcock and P.G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 73–82. Springer-Verlag, 1993.

[HG94] R. Herzig and M. Gogolla. An Animator for the Object Specification Language TROLL *light*. In *62ème Congrés de l'Association Canadienne Française pour l'Avancement des Sciences (ACFAS): Orientation Object en Bases de Données et Génie du Logiciel*, pages 4–17, Montreal, Canada, May 1994.

[HJ89] I.J. Hayes and C.B. Jones. Specifications are not (Necessarily) Executable. *Software Engineering Journal*, 4(6):330–338, November 1989.

[HJS92] T. Hartmann, R. Jungclaus, and G. Saake. Aggregation in a Behavior Oriented Object Model. In O.L. Madsen, editor, *ECOOP'92*, volume 615 of *Lecture Notes in Computer Science*, pages 57–77. Springer-Verlag, 1992.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hoa94] R. Hoare, C.A. How Did Software Get so Reliable Without Proof? BCS Proof Club, Edinburgh, March 1994.

[HS92] B. Henderson-Sellers. Object-Oriented Information Systems: An Introductory Tutorial. *The Australian Computer Journal*, 24(1):12–24, February 1992.

[HS93] G.M. Høydalsvik and G. Sindre. On the Purpose of Object-Oriented Analysis. In *Proceedings of OOPSLA'93: ACM SIGPLAN Notices*, 28(10):240-258, October 1993.

[Hür94] W.L. Hürsch. Should Superclasses be Abstract? In M. Tokoro and R. Pareschi, editors, *ECOOP'94*, volume 821 of *Lecture Notes in Computer Science*, pages 12–31. Springer-Verlag, 1994.

[IEE91] IEEE. IEEE Standard Glossary of Software Engineering Terminology. Institute of Electrical and Electronic Engineers, Inc., New York, USA, 1991. Revision and redesignation of IEEE Std 729-1983. Reprinted in *IEEE Software Engineering Standards Collection*, 1993.

[Ier93] R. Ierusalimschy. A Formal Specification for a Hierarchy of Collections. *Software Engineering Journal*, 8(4):237–244, July 1993.

[IH87] D.C. Ince and S. Hekmatpour. Software Prototyping – Progress and Prospects. *Information and Software Technology*, 29(1):8–14, January/February 1987.

[ISO88] ISO. Information Processing Systems – Open Systems Interconnection – LOTOS : A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, International Standard 8807. ISO, 1988.

[ISO94] ISO. Basic Reference Model of Open Distributed Processing. ISO/IEC JTC1/SC21, American National Standards Institute, 1994. Draft standard.

[Jac92] I. Jacobson. *Object-Oriented Software Engineering — A Use Case Driven Approach*. Addison-Wesley, 1992.

[JDP89] L. Jadoul, L. Duponcheel, and W. Van Puymbroeck. An Algebraic Data Type Specification Language and its Rapid Prototyping Environment. In *11th International Conference on Software Engineering*, IEEE Computer Society Press, pages 74–42, Pittsburgh, Pennsylvania, 1989.

[Jon80] C.B. Jones. *Software Development. A Rigorous Approach*. Prentice-Hall, 1980.

[Jon86] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1986.

[Jon93] C.B. Jones. A Pi-Calculus Semantics for an Object-Based Design Notation. In E. Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, 1993.

[JSHS91] R. Jungclaus, G. Saake, R. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Technical Report 91-04, Technische Universität Braunschweig, Informatik Berichte, Postfach 3329, W-3300 Braunschweig, Germany, December 1991.

[KBC+87] W. Kim, J. Banerjee, H. Chow, J.F. Garza, and D. Woelk. Composite Object Support in an Object-Oriented Database System. In *Proceedings of OOPSLA'87: ACM SIGPLAN Notices*, 22(12):118-125, December 1987.

[Lad94] P. Ladkin. Deriving the Behaviour of Objects: Personal Communication, 1994.

[Lam93] L. Lamport. The Temporal Logic of Actions. Technical Report 79, Digital Equipment Corporation, Systems Research Center, November 1993.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 1994. To appear.

[Li93] W. Li. A Theory of Requirements Capture and Its Applications. In M.C. Gaudel and J.P. Jouannaud, editors, *TAPSOFT'93*, volume 668 of *Lecture Notes in Computer Science*, pages 406–420. Springer-Verlag, 1993.

[Lim93] P. Lim. Applying Rigorous Object-Oriented Analysis. Master's thesis, Department of Computing Science and Mathematics, University of Stirling, Scotland, September 1993.

[LL94] P.B. Ladkin and S. Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, 1994. To appear.

[LS83] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[LS93] A. Laorakpong and M. Saeki. Object-Oriented Formal Specification Using VDM. In S. Hishio and A. Yonezawa, editors, *Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 529–543. Springer-Verlag, 1993.

[May89] T. Mayr. Specification of Object-Oriented Systems in LOTOS. In K.J. Turner, editor, *Formal Description Techniques*, pages 107–119. North-Holland, 1989.

[MC83] R. Mason and T. Carey. Prototyping Iterative Information Systems. *Communications of the ACM*, 28(5):347–354, May 1983.

[MC92] A.M.D. Moreira and R.G. Clark. Object-Oriented Analysis and its Relation to Object-Oriented Design. Technical Report CSM-089, Department of Computing Science and Mathematics, University of Stirling, Scotland, May 1992.

[MC93a] A.M.D. Moreira and R.G. Clark. LOTOS in the Object-Oriented Analysis Process. In *BCS-FACS Workshop on Formal Aspects of Object-Oriented Systems*, Imperial College, London, December 1993. *BCS-FACS (British Computer Society – Formal Aspects of Computing Science).*

[MC93b] A.M.D. Moreira and R.G. Clark. Os Métodos Formais na Análise de Orientação por Objectos. In *7th Brazilian Symposium on Software Engineering*, Rio de Janeiro, Brazil, pages 238–252, October 1993. An English version of this paper is presented in the Technical Report CSM-111, Department of Computing Science and Mathematics, University of Stirling.

[MC93c] A.M.D. Moreira and R.G. Clark. ROOA: Rigorous Object-Oriented Analysis. Technical Report CSM-109, Department of Computing Science and Mathematics, University of Stirling, Scotland, October 1993.

[MC93d] A.M.D. Moreira and R.G. Clark. Using Rigorous Object-Oriented Analysis. Technical Report CSM-111, Department of Computing Science and Mathematics, University of Stirling, Scotland, August 1993. *Presented at the 7th Brazilian Symposium on Software Engineering, October 1993.*

[MC94a] A.M.D. Moreira and R.G. Clark. Combining Object-Oriented Analysis and Formal Description Techniques. In M. Tokoro and R. Pareschi, editors, *ECOOP'94*, volume 821 of *Lecture Notes in Computer Science*, pages 344–364. Springer-Verlag, 1994.

[MC94b] A.M.D. Moreira and R.G. Clark. Complex Objects: Aggregates. Technical Report CSM-123, Department of Computing Science and Mathematics, University of Stirling, Scotland, May 1994.

[MC94c] A.M.D. Moreira and R.G. Clark. O Método ROOA. In A. Vaz-Velho and P.G. Guedes, editors, *Object-Oriented Portugal (OOP'94)*, pages 67–76, Lisbon, Portugal, September 1994.

[MC94d] A.M.D. Moreira and R.G. Clark. Rigorous Object-Oriented Analysis. In E. Bertino and S. Urban, editors, *International Symposium on Object-Oriented Methodologies and Systems (ISOOMS)*, volume 858 of *Lecture Notes in Computer Science*, pages 65–78. Springer-Verlag, 1994.

[MC94e] A.M.D. Moreira and R.G. Clark. Specification Case Studies in ROOA. Technical Report CSM-129, Department of Computing Science and Mathematics, University of Stirling, Scotland, October 1994.

[McC93] A. McClenaghan. *Distributed Systems: Architecture-Driven Specification Using Extended LOTOS.* PhD thesis, Department of Computing Science and Mathematics, University of Stirling, Scotland, September 1993.

[MFG89] A.M.D. Moreira, M.M. Freitas, and P. Guerreiro. Using Object Oriented Requirements Analysis for a High Reliability Message Switching System. In *AFCEA Portugal*, Lisbon, Portugal, May 1989.

[MGF90] A.M.D. Moreira, P. Guerreiro, and M.M. Freitas. Métodos de Análise de Requisitos Orientada pelos Objectos. In *VI Congresso Português de Informática*, Lisbon, Portugal, June 1990.

[Mil56] G.A. Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review*, 63(2):81–97, March 1956. Reprinted in Yourdon *Writings of the Evolution*, pages 443-460, 1982.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[MLC94] A.M.D. Moreira, P.B. Ladkin, and R.G. Clark. Formalizing OO Analysis with LOTOS. Technical Report CSM-125, Department of Computing Science and Mathematics, University of Stirling, Scotland, August 1994.

[NW93] F. Nickl and M. Wirsing. A Formal Approach to Requirements Engineering. In D. Bjørner and M. Broy, editors, *Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 312–334. Springer-Verlag, 1993.

[Ode94] J. Odell. Six Different Kinds of Composition. *Journal of Object-Oriented Programming*, 6(8):10–15, January 1994.

[Par72] D.L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 5(12):1053–1058, December 1972.

[PC86] D.L. Parnas and P.C. Clements. A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, February 1986.

[Pet88] L.J. Peters. *Advanced Structured Analysis and Design*. Prentice-Hall, 1988.

[PKT92] N. Plat, J. Katwijk, and H. Toetenel. Application and Benefits of Formal Methods in Software Development. *Software Engineering Journal*, 7(5):335–346, September 1992.

[Por92] H. H. Porter. Separating the Subtype Hierarchy from the Inheritance of Implementation. *Journal of Object-Oriented Programming*, 4(9):20–29, February 1992.

[Rat88] B. Ratcliff. Early and Not-so-Early Prototyping – Rationale and Tool Support. In *12th Annual International Computer Software and Applications Conference, COMPSAC'88*, IEEE Computer Society Press, pages 127–134, Chicago, October 1988.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall, 1991.

[RG92] K.S. Rubin and A. Goldberg. Object Behaviour Analysis. *Communications of the ACM*, 35(9):48–62, September 1992.

[Roy70] W.W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *WESCON*, San Francisco CA, August 1970.

[RS88] J. Reif and S.A. Smolka. The Complexity of Reachability in Distributed Communicating Processes. *Acta Informatica*, 25:333–354, 1988.

[Rud92] S. Rudkin. Inheritance in LOTOS. In K.R. Parker and G.A. Rose, editors, *Formal Description Techniques, IV*, pages 409–423. North-Holland, 1992.

[Rud93] S. Rudkin. Templates, Types and Classes in Open Distributed Processing. *BT Technology Journal*, 11(3):32–40, July 1993.

[SB82] W. Swartout and R.M. Balzer. The Inevitable Intertwining of Specification and Implementation. *Communications of the ACM*, 25(7):438–440, July 1982.

[SBC92] S. Stepney, R. Barden, and D. (editors) Cooper. *Object Orientation in Z.* Springer, 1992.

[Shu91] K. Shumate. Structured Analysis and Object-Oriented Design are Compatible. *Ada Letters*, 11(4):78–90, May/June 1991.

[SM89] S. Shlaer and S.J. Mellor. An Object-Oriented Approach to Domain Analysis. *ACM Software Engineering Notes*, 14(5):66–77, July 1989.

[SM92] S. Shlaer and S.J. Mellor. *Object Lifecycles — Modeling the World in States.* Prentice-Hall, 1992.

[Smi91] M.F. Smith. *Software Prototyping — Adoption, Practice and Management.* McGraw-Hill, 1991.

[Som92] I. Sommerville. *Software Engineering.* Addison-Wesley, 4th edition, 1992.

[Spi89] J.M. Spivey. *The Z Notation.* Prentice-Hall International, 1989.

[SS77] J.M. Smith and D.C.P. Smith. Database Abstractions: Aggregation. *Communications of the ACM*, 20(6):405–413, June 1977.

[Sta93] M. Stark. Impacts of Object-Oriented Technologies: Seven Years of SEL Studies. In *Proceedings of OOPSLA'93: ACM SIGPLAN Notices*, 28(10):365-373, October 1993.

[Tur93] K.J. Turner, editor. *Using Formal Description Techniques.* John Wiley & Sons, 1993.

[TY92] S. Tysyberowicz and A. Yehudai. OBSERV–A Prototyping Language and Environment. *ACM Transactions on Software Engineering and Methodology*, 1(3):269–309, July 1992.

[vEVD89] P.H. van Eijk, C.A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS: Results of the ESPRIT/SEDOS Project.* North Holland, 1989.

[Wal91] N.L. Walters. An Ada Object-Based Analysis and Design Approach. *Ada Letters*, 11(5):62–78, July/August 1991.

[War89] P. Ward. How to Integrate Object Orientation with Structured Analysis and Design. *IEEE Software*, 6(2):74–82, March 1989.

[WBWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[Weg87] P. Wegner. Dimensions of Object-Based Language Design. *Special Issue of SIGPLAN Notices*, 22(12):168–182, October 1987.

[WRW93] W. Wessale, D. Reifer, and D. Weller. Large Project Experiences with Object-Oriented Methods and Reuse. *The Journal of Systems and Software*, 23(2):151–161, November 1993.

[You82] E. Yourdon. *Writings of the Evolution*. Yourdon Press, Prentice-Hall, 1982.

[You89] E. Yourdon. *Modern Structured Analysis*. Prentice-Hall, 1989.

[Zav84] P. Zave. The Operational versus the Conventional Approach to Software Development. *Communications of the ACM*, 27(2):104–118, February 1984.

[Zav91] P. Zave. An Insider's Evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 17(3):212–225, March 1991.

[Zel80] M. Zelkowitz. A Case Study in Rapid Prototyping. *Software – Practice and Experience*, 10(2):1037–1042, December 1980.

[ZY81] P. Zave and R.T. Yeh. Executable Requirements for Embedded Systems. In *5th International Conference on Software Engineering*, IEEE Computer Society Press, pages 295–304, Washington, D.C., 1981.

Appendix A

LOTOS Overview

A.1 Introduction

LOTOS is a formal description technique developed by ISO for the definition of Open Systems Interconnection (OSI) standards, although it is also well suited to the specification of a wide range of systems, including embedded systems [Cla92b]. It has two main components:

- Process definition.
  This component describes the behaviour of processes and the interactions between them. The approach used is based on process algebra, using components from CCS [Mil89] and CSP [Hoa85].
- Abstract data types.
  This component describes the data types and value expressions. It is based on the abstract data type language ACT ONE [EM85].

A.2 Processes

A concurrent distributed system is described in LOTOS as a set of communicating processes. A process is considered to be a black box and its externally observable behaviour is its interactions with other processes. Specifying a process is defining the temporal relationships among such interactions.

Process behaviour is described using *behaviour expressions* that consist of external, observable events and internal, externally unobservable events. Processes are composed by using the parallel operators and they interact with each other through synchronization on *events*. An event is atomic and takes place at an *event gate* (or just *gate*). It appears in a process definition and is composed of a gate name followed by a set of arguments in which the operator "!" is used in the form `!v` where `v` is a value expression, and the operator "?" is used in the form `?x: s` where `x` is a variable of the sort `s`.

For example, in the event:

```
gate_name !val ?x: Nat
```

the term `!val` indicates that the value `val` is to be transmitted and the term `?x: Nat` indicates that any value of the sort `Nat` can be accepted and assigned to `x`.

There are restricted conditions in which events synchronize. The event:

```
gate_name !val !num
```

would synchronize with the event above if `!num` is some value of sort `Nat`.

Table A.1 summarizes the three types of synchronization [BB87].

Process A	**Process B**	**Condition of Synchronization**	**Interaction Type**	**Effect**
g $!E_1$	g $!E_2$	value(E_1) = value(E_2)	value matching	synchronization occurs
g $!E_1$	g ?x: s	sort(E_1) = s	value passing	after synchronization x = value(E_1)
g ?y: w	g ?x: s	w = s	value generation	after synchronization y = x = v, where v is some value of sort w

Table A.1: Interaction types

Value matching is used to ensure synchronization is achieved. Value passing is used to pass a value to a variable. Value generation allows the introduction of uninstantiated variables.

A process definition has the following syntax:

process process_name [list of gates](list of parameters) : functionality :=
    (* behaviour expression *)
where
    (* data type definitions *)
    (* process definitions *)
endproc

The functionality can be: **exit**, meaning that the process may terminate successfully, **noexit**, meaning that the process cannot terminate (perhaps because it recursively calls itself) and **exit** *(result)*, meaning that the process may terminate successfully and return a result. ("(*" begins a comment and "*)" ends it.)

The body of the process defines its behaviour in terms of its process components (if any) and the events in which it can take part. We can also define abstract data types within a process definition.

Here is a simple example of a process that offers a value greater than the value received as a parameter:

```
process GreaterValue[g](count: Nat) : noexit :=
  g ?ncount: Nat [ncount gt count];
  GreaterValue[g](ncount)
endproc
```

The process `GreaterValue` is defined recursively and uses gate `g` for synchronization with other processes.

The behaviour expression:

```
g ?ncount: Nat [ncount gt count];
```

offers a value of sort `Nat` for synchronization. The selection predicate `[ncount gt count]` guarantees that only values greater than the value offered in the previous instantiation are allowed.

A.3 Abstract Data Types

LOTOS represents data as abstract data types (ADTs) using the language ACT ONE. An ADT definition is rather lengthy and complex although this can be made easier by the provision of an extensive library of predefined ADTs.

The structure used to define a type is always the same, with the sections in the following order (some of these sections are optional):

type type_name **is**
  (* list of imported definitions *)
  sorts sort_name
  opns
    (* list of operations *)
  eqns forall
    (* list of variables *)
    ofsort a_sort_name
      (* list of equations *)
    ofsort a_sort_name
      (* list of equations *)
    ...
endtype

The **type** section gives the name of the definition (this is the name that should be used to combine different definitions). A list of imported definitions can appear after the keyword **is**. The **sorts** section gives the name of the data sorts. The **opns** section gives the signature of the operations. An operation is a function with zero or more sorts as its domain and with only one sort as its codomain. The **eqns** section specifies, in terms of equations, the constraints the operations must satisfy. This section uses the keyword **eqns forall** after which we declare the variables that are going to be used in the equations, and the keyword **ofsort** after which we define the result sort of the equations and then the equations themselves. Because different equations can have different result sorts, the latter keyword can appear repeatedly.

The next example defines a stack of natural numbers as a LOTOS ADT:

```
type Stack_Type is NaturalNumber, Boolean
  sorts Stack
  opns EmptyStack :             -> Stack
       Push       : Stack, Nat -> Stack
       Pop        : Stack      -> Nat
       IsEmpty    : Stack      -> Bool
  eqns forall s: Stack, n1: Nat
    ofsort Nat
```

```
        Pop(Push(s, n1))  = n1;
        Pop(EmptyStack)   = 0;
     ofsort Bool
        IsEmpty(EmptyStack)  = true;
        IsEmpty(Push(s, n1)) = false;
endproc
```

The operations `Push` and `EmptyStack` are constructors, i.e. they create a value of the ADT, and they do not have any defining equations. The constant "0" has been defined to be of sort `Nat` in type `NaturalNumber`. It is used here to indicate an error when the `Pop` operation is applied to an empty stack. This is only an example, and should not be considered the best definition of a stack.

A.4 Overall Structure of a LOTOS Specification

A general LOTOS specification has the following structure:

specification specification_name [list of gates](list of parameters) : functionality
(* data type definitions *)
library ... **endlib**
type ... **endtype**
type ... **endtype**
...
(* process definitions *)
behaviour
(* behaviour expression *)
where
process ... **endproc**
process ... **endproc**
...
endspec

In the library are defined the commonly used data types that can be used either directly or in the construction of more complex data types. In the behaviour part it is possible to have both abstract data type definitions and nested process definitions. However, the option of defining ADTs inside a process is not much used, because ADT definitions are often long and it would make the process more difficult to understand.

Table A.2 summarizes the syntax of the most common behaviour expressions. These can be combined to define complex behaviour expressions.

For further reading on the LOTOS language, see for example [BB87, Tur93].

Name	**Syntax**
inaction	**stop**
termination	**exit**
termination with parameters	**exit**(E_1,...,E_n)
choice	B_1 [] B_2
generalized-choice	**choice** v:T [] B
	choice g **in** [g_1, ..., g_n] [] B
action-prefix:	
observable (external)	g;B
observable with selection predicate	$g\ d_1 \ldots d_n[CE]$;B
unobservable (internal)	**i**;B
parallel composition:	
general case	B_1 |[g_1, ..., g_n]| B_2
interleaving	B_1 ||| B_2
full synchronization	B_1 || B_2
hiding	**hide** g_1,...,g_n **in** B
process instantiation	P[g_1,...,g_n] ($E_1 \ldots E_n$)
guarding	[CE] -> B
disabling	B_1 [> B_2
enabling	B_1 >> B_2
enabling with value passing	B_1 >> **accept** v_1: T_1,...,v_n: T_n **in** B_2
local definition	**let** v_1: T_1 = E_1,...,v_n: T_n = E_n **in** B

Legend:

B, B_1, B_2 : behaviour expressions	T_1,..., T_n : sort identifiers
v_1,..., v_n : variable identifiers	E_1,..., E_n : value expressions
g_1,..., g_n : gate identifiers	CE : conditional expression
d_1,..., d_n : experiment offers	P : process identifier

Table A.2: Syntax of the most important LOTOS operators

Appendix B

Additions to the LOTOS Libraries

Although LOTOS comes with libraries where some data types are defined, we must specify almost all the data types we need. Most of these data types are defined based on the ones exported by the library. Each object needs an identifier to allow it to be referenced by other objects or by the external world. Instead of defining from scratch an identifier for each object, we have included in the library the abstract data types `Id Type` and `Set Id Type` to be used as a starting point.

B.1 Defining a Type Identifier

`Id_Type` is an identifier definition and we specify it as follows:

```
type Id_Type is Boolean, NaturalNumber
   sorts Id
   opns id1, id2, id3, id4, id5, id6, id7, id8, id9, id10, id11,
        id12, id13, id14, id15, id16, id17, id18, id19, id20 : -> Id
        _eq_, _ne_, _lt_ : Id, Id -> Bool
        h           : Id -> Nat
        First_Set   : Id -> Bool
        Second_Set  : Id -> Bool
        Third_Set   : Id -> Bool
        Fourth_Set  : Id -> Bool
   eqns forall n1, n2: Id
     ofsort Nat
       h(id1) = 0;
       h(id2) = succ(h(id1));
       h(id3) = succ(h(id2));
       h(id4) = succ(h(id3));
       h(id5) = succ(h(id4));
       h(id6) = succ(h(id5));
       h(id7) = succ(h(id6));
       h(id8) = succ(h(id7));
       h(id9) = succ(h(id8));
       h(id10) = succ(h(id9));
       h(id12) = succ(h(id11));
```

```
      h(id13) = succ(h(id12));
      h(id14) = succ(h(id13));
      h(id15) = succ(h(id14));
      h(id16) = succ(h(id15));
      h(id17) = succ(h(id16));
      h(id18) = succ(h(id17));
      h(id19) = succ(h(id18));
      h(id20) = succ(h(id19));
    ofsort Bool
      n1 eq n2 = h(n1) eq h(n2);
      n1 ne n2 = h(n1) ne h(n2);
      n1 lt n2 = h(n1) lt h(n2);
      First_Set(n1)  = h(n1) lt h(id5);
      Second_Set(n1) = not(h(n1) lt h(id5))  and (h(n1) lt h(id10));
      Third_Set(n1)  = not(h(n1) lt h(id10)) and (h(n1) lt h(id15));
      Fourth_Set(n1) = not(h(n1) lt h(id15)) and (h(n1) lt h(id20));
endtype
```

We have defined 20 different identifiers, but we can define as many as we want. We also define the operations `First_Set`, `Second_Set`, `Third_Set` and `Fourth_Set` in order to allow different classes of objects to share the same sort of identifiers. This is required to specify subclasses (see Section 3.3.9).

B.2 Defining Sets of Identifiers

In a normal situation we need to be able to create multiple objects of the same class template. In order to accomplish that we specify the ADT `Set_Id_Type` which permits us to define sets of `Id_Type`. The standard LOTOS libraries already include the parameterized ADT `Set` for sets. `Set` has two formal parameters, `Element`, which is actualized with `Id`, and `FBool` which is actualized with `Bool`. (`Bool` is a sort for booleans defined in the LOTOS libraries.)

Therefore, the definition of `Set_Id_Type` is an actualization of the `Set` ADT, as follows:

```
type Set_Id_Type is Set actualizedby Id_Type using
    sortnames Id   for Element
              Bool for FBool
endtype
```

With these two ADTs added to the library, we can define new abstract data types for sets of object identifiers by renaming `Set_Id_Type`, using `Set` type and `Id_Type` as arguments.

For example, for the banking system, we can define `Account_Number_Set_Type`:

```
type Account_Number_Set_Type is Set_Id_Type
   renamedby
   sortnames Account_Number     for Id
             Account_Number_Set for Set
endtype
```

This defines the sort `Account_Number` and the sort `Account_Number_Set`.

Appendix C

Publications

We started working with object-oriented and object-based methods for analysis and design in 1988 and with formal specification languages in 1991. Here we present a list of our publications on these subjects. (The full references including co-authors are given in the Bibliography.)

C.1 Articles in Conferences and Journals

- *Combining Object-Oriented Analysis and Formal Description Techniques*, 8th European Conference on Object-Oriented Programming, ECOOP'94, Lecture Notes in Computer Science, volume 821, Bologna , Italy, July 1994.
- *Rigorous Object-Oriented Analysis*, International Symposium on Object-Oriented Methodologies and Systems (ISOOMS), Lecture Notes in Computer Science, volume 858, Palermo, Italy, September 1994.
- *O Método ROOA*, Object-Oriented Portugal, OOP'94, Lisbon, Portugal, September 1994.
- *Behavioural Inheritance in ROOA*, 4th Workshop on Information Systems - Correctness and reusability (IS-CORE'94), Amsterdam, The Netherlands, September 1994.
- *Os Métodos Formais na Análise de Orientação por Objectos*, 7th Brazilian Symposium on Software Engineering, Rio de Janeiro, Brazil, October 1993.
- *LOTOS in the Object-Oriented Analysis Process*, BCS-FACS Workshop on Formal Aspects of Object-Oriented Systems, December 1993.
- *Introducing Object-Oriented Methodologies with Ada in Portugal*, International Conference of Asia Pacific Defense'90, Seoul, Korea, November 1990.
- *Object-Oriented Requirements Analysis in an Ada Project*, Ada Letters, 10(6), July/ August 1990.
- *Métodos de Análise de Requisitos Orientada pelos Objectos*, VI Congresso Português de Informática, Lisbon, Portugal, June 1990.

- *Choosing Ada Development Methodologies for a High Reliability Message Switching System*, 4th AFCEA Hawaii Defense Electronics, Hawaii, December 1989.
- *Using Object-Oriented Design with Ada for a High Reliability Message Switching System*, AFCEA International's Asia Pacific Defense'89, Seoul, Korea, September 1989.
- *Using Object Oriented Requirements Analysis for a High Reliability Message Switching System*, AFCEA Portugal, Lisbon, Portugal, May 1989.

C.2 Technical Reports

- *Specification Case Studies in ROOA*, Department of Computing Science and Mathematics, University of Stirling, CSM-129, Scotland, October 1994.
- *Formalizing OO Analysis with LOTOS*, Department of Computing Science and Mathematics, University of Stirling, CSM-125, Scotland, August 1994.
- *Complex Objects: Aggregates*, Department of Computing Science and Mathematics, University of Stirling, CSM-123, Scotland, May 1994.
- *ROOA: Rigorous Object-Oriented Analysis*, Department of Computing Science and Mathematics, University of Stirling, CSM-109, Scotland, October 1993.
- *Using Rigorous Object-Oriented Analysis*, Department of Computing Science and Mathematics, University of Stirling, CSM-111, Scotland, August 1993.
- *Object-Oriented Analysis and its Relation to Object-Oriented Design*, Department of Computing Science and Mathematics, University of Stirling, CSM-089, Scotland, May 1992.

Appendix D

Acronyms

This appendix contains a list of acronyms used in the thesis.

Acronyms	Expansion
ADT	Abstract Data Type
CASE	Computer Aided Software Engineering
CCS	Calculus of Communicating Systems
CSP	Communicating Sequential Processes
DFD	Data Flow Diagram
ERD	Entity-Relationship Diagram
ETD	Event Trace Diagram
FDT	Formal Description Technique
ISO	International Organisation for Standardisation
LITE	LOTOSPHERE Integrated Tool Environment
LOTOS	Language Of Temporal Ordering Specification
OCD	Object Communication Diagram
OCT	Object Communication Table
ODP	Open Distributed Processing
OMT	Object Modelling Technique
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OOSE	Object-Oriented Software Engineering
OSI	Open Systems Interconnection
ROOA	Rigorous Object-Oriented Analysis
SEDOS	Software Environment for the Design of Open Distributed Systems
TLA	Temporal Logic of Actions