Using Rigorous Object-Oriented Analysis

Ana M D Moreira * and Robert G Clark
Department of Computing Science and Mathematics,
University Of Stirling,
STIRLING FK9 4LA, Scotland.
Email: amm@uk.ac.stir.cs
Email: rgc@uk.ac.stir.cs

A Portuguese version of this paper is to be presented at the 7th Brazilian Symposium on Software Engineering on 26-29 October 1993, in Rio de Janeiro, Brazil.

Abstract

The ROOA (Rigorous Object-Oriented Analysis) method introduces formality into the object-oriented analysis process by providing a set of rules which enables a formal object-oriented analysis model to be produced systematically from a set of requirements. This model is expressed in LOTOS and provides a precise and unambiguous specification of system requirements. As the specification obtained is executable, prototyping is used to support validation and refinement of the formal model.

*Supported by the Junta Nacional de Investigação Cientifica e Tecnológica (JNICT), Portugal.

Contents

1 Introduction

Developing an efficient, reliable and maintainable software system requires the adoption of a strategy that helps software engineers to communicate without ambiguity. Designers must be able to understand the results provided by analysts and give an unambiguous specification to the implementors. A solution is to provide a rigorous software development process which includes the development of a formal requirements specification so that the requirements can be stated precisely and unambiguously.

In this paper we introduce the ROOA (Rigorous Object-Oriented Analysis) method [15] by means of an example. ROOA takes the static properties captured in an object model produced by any object-oriented analysis method [5, 12, 16, 18] together with the dynamic and functional properties given in the original requirements and produces an executable formal object-oriented requirements specification. Formal Description Techniques, such as [1, 3, 13], are usually applied after the requirements analysis phase, but here we are using them to help in determining and understanding a system's requirements. The formal description technique we have chosen is LOTOS (Language Of Temporal Ordering Specification) [1] which has a precise formal mathematical semantics and which can be used in an object-oriented style. The resulting formal model considers the system as a set of concurrent objects where message passing is modelled by objects synchronizing on an event during which information may be exchanged. The specification gives an integrated description of the system which deals with both static and dynamic properties. In other methods these properties are normally described by different techniques which leads to problems in ensuring that the different descriptions remain consistent as the model is developed.

As the specification obtained is executable, prototyping can be used for validation and to check the conformance of different specifications produced during a refinement process. By combining the use of formal description techniques with rapid prototyping during analysis we can discover inconsistencies, omissions, contradictions and ambiguities early, so that they can be corrected in the early stages of the development process. The formal requirements specification can subsequently be transformed into a formal design specification. Prototyping with the same set of interface scenarios can be used to check that the observable behaviour of the design specification conforms to that of the requirements specification.

Section 2 discusses the need for formal specifications. Section 3 gives a introduction to object-oriented analysis methods. Section 4 summarizes the ROOA method. Section 5 introduces LOTOS. Section 6 shows how the ROOA method can be used to derive a formal LOTOS object-oriented model. Finally, Section 7 gives our conclusions.

2 Formal and Executable Specifications

The primary benefit of formal techniques is that, as they have a precise and mathematical semantics, the resulting specifications are unambiguous. This is in contrast to informal techniques which 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 lead to errors and omissions in the code, resulting in high costs for support and repair. Moreover, this imprecision leads to misunderstandings in validating the informal specification against the requirements (and the implementation against the specification). A formal approach to specification is therefore required. A formal requirements specification, at least in theory, allows an implementation to be verified against the specification, although it still leaves the problem of

validating the specification against the initial informal requirements document.

Proving that a requirements specification, 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.

Not all software engineers agree that specifications should be executable, because a specification written in a notation that is not directly executable will contain less implementation detail than an executable one [9]. There is also the danger that executable specifications can overspecify a problem. Being able to demonstrate that a specification exhibits the expected behaviour can, however, greatly increase ones confidence in it [8]. The accusation that this is no more than testing, is partially solved by using symbolic evaluation. The LOTOS SMILE simulator [6] allows the use of uninstantiated variables within conditions and is able to determine when a combination of conditions can never be true. Many more behaviours can then be examined with each simulation than is possible when all data values have to be instantiated.

3 Object-Oriented Analysis Methods

The main goal of an object-oriented analysis method is to identify objects and classes which constitute a system, to understand the structure and behaviour of each object, to gather in one place (localization) 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 the user requirements.
2. Identify and classify objects.
3. Define objects.
4. Identify relationships between objects.
5. Construct documentation.

Understanding the user requirements is accomplished by reading the initial requirements document and any other source of information where the problem, or part of it, may be described. The users or clients of the system should also be interviewed.

To identify and classify objects, several methods [5, 18] suggest we look at nouns, pronouns, noun phrases, adjectival and adverbial phrases in the initial requirements document, while others [16] suggests that a better way to identify objects is by identifying their behaviour in the system.

An object is defined in terms of its static and its dynamic aspects. The static aspect is given by a list of its attributes and operations. The dynamic behaviour is usually described by using state transition diagrams, but it plays a secondary role in most of the 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 and 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 where enhancements have been introduced to support aggregates, inheritance and message connections. Some methods [18] add message sequence charts to the dynamic model to show the interactions between objects.

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

More recent methods, such as [18, 19], also incorporate a *functional model* which uses data flow diagrams to describe the meaning of the operations in the object model and the actions in the dynamic model.

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 not difficult. Moreover, the techniques used by the object-oriented design methods usually produce designs which are very close to code. Sometimes they already are outline code, as when Ada or Eiffel is used as a design language.

4 The Rigorous Object-Oriented Analysis Method

The ROOA (Rigorous Object-Oriented Analysis) method involves three main tasks. In the first task we build an object model. In the second task we refine the object model by normalizing it and by identifying object hierarchies. In the third task we describe a set of subtasks that should be followed in building the formal LOTOS object-oriented analysis model.

ROOA follows a parallel/recursive approach which allows us to re-apply the whole (or part) of the method to the results of a previous iteration. Moreover, we can apply the method to different parts of the system (subsystems) at the same time (in parallel). In each refinement we introduce more detail to the formal model.

Task 1: Build the Object Model

Before we start producing the formal model, we have to build an object model by using any of the standard object-oriented analysis methods [5, 12, 16, 18]. The construction of the initial object model can be considered as a completely separate task from the following ones and it can be accomplished by a different team. During the application of our method, the object model may be modified.

An advantage of starting with an object model produced by any object-oriented analysis method is that we build on the work which has already been done to identify objects.

Task 2: Refine the Object Model

In this task we *normalize* the object model, i.e. we guarantee that it has static relationships, attributes, operations, and message connections. We also identify hierarchies of objects, i.e. higher level objects, in order to make the system easier to understand and develop. This task is difficult and so we cannot expect to do it completely and correctly in the first iteration. The low level objects in the object model often remain almost unchanged during the development, but the high level structure is less stable. We only identify obvious hierarchies (aggregates and inheritance) to begin with and then, as the approach is recursive/parallel, come back to it when our knowledge about each individual object increases.

Task 3: Build the LOTOS Formal Model

During this task we create an object communication diagram, specify objects and classes, compose the objects into LOTOS behaviour expressions, prototype and refine the specification.

1. Create an object communication diagram.
   This diagram is a graph where, in the first iteration, a node represents an object and each arc connecting two objects represents a gate of communication between them. In later iterations the diagram will be generalised to deal with multiple objects of the same class. 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 be connected to the others and new groupings will appear, refining the diagram.

   We can determine, by examining the message connections in the object model, whether an object acts as a client, server, or a combination of both. If it is a server, then it offers services to its different clients at one gate.

2. Specify individual objects and classes.
   In general, the behaviour of an object is specified as a process and its state information as one or more Abstract Data Types (ADTs) given as parameters of the process. For each individual object we:

   (a) Start specifying the process by identifying the events it takes part in and their order.

   (b) Start specifying symbolic ADTs to describe its attributes.

   Inheritance in LOTOS is more of a problem and a theoretical study has been made by Rudkin [17]. There are two main definitions of inheritance [11]. In *behavioural inheritance*, objects of a subclass offer all the services of objects of their superclass and can be used wherever an object of the superclass is expected. In *incremental inheritance*, a subclass inherits the definition of its superclass which it then extends.

   We believe that, in a specification, the behavioural and incremental inheritance hierarchies should be restricted to be the same. Although LOTOS does not directly support inheritance, it is straightforward to represent incremental inheritance and examples are given in Section 6. Multiple inheritance is not supported.

3. Compose the objects into a behaviour expression.
   Following the structure of the object communication diagram we compose the processes into a LOTOS behaviour expression by using the LOTOS parallel operators.

4. Prototype the specification.
   We use interface scenarios and prototyping to check services and message connections. Any errors, omissions or inconsistencies found will lead us to go back to one or more tasks and update the object model, the object communication diagram and the specification.

5. Refine the specification.
   The specification is refined by performing the whole process in a recursive/parallel approach. During successive refinements we may identify new higher level objects, define object generators so that multiple instances of the same class can be created dynamically, demote an object to be specified only as an ADT, promote an object so that it requires a process and refine processes and ADTs by introducing more detail.

4.1 Automated Banking System

The problem we have chosen to show how to use our method 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. Saving accounts give interest and cannot be accessed by the automatic tellers.

We applied the object-oriented analysis methods of OOA [5] and OMT [18] to this problem, but only the final object model produced by [18] is presented here using the notation of OMT (see Figure 1). This object model shows the class of objects with attributes and the relationships between objects.

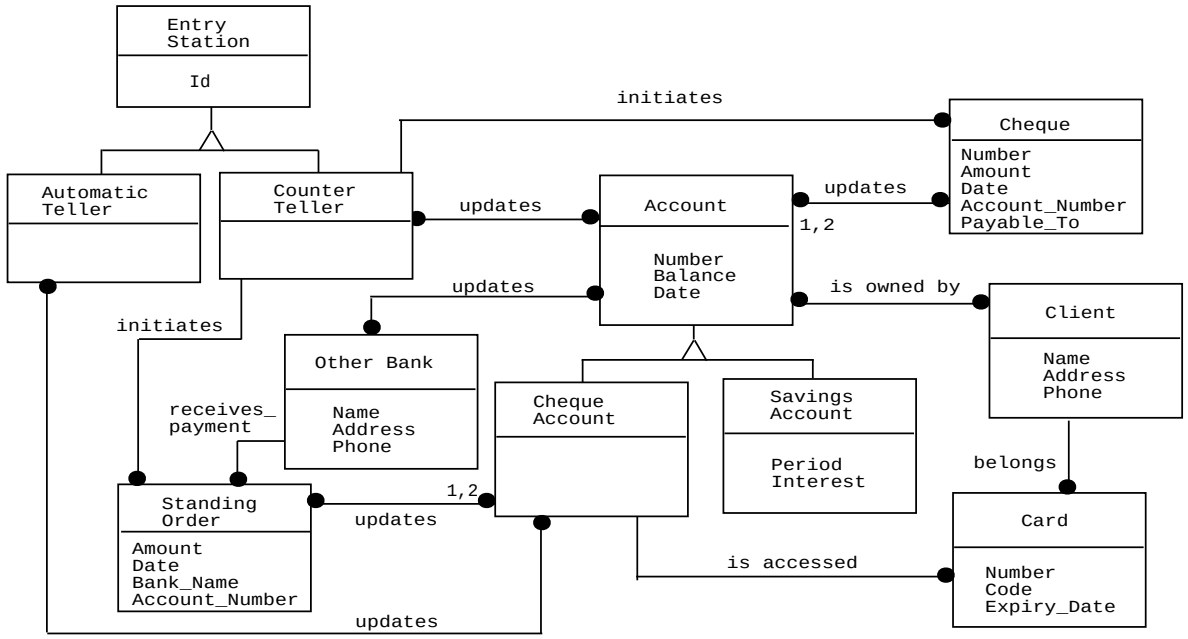

Figure 1: Object model produced by the OMT method

5 LOTOS Overview

LOTOS is a formal description technique developed by ISO [2] 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 [4]. It has two main components:

- Process definition: this component describes the behaviour of processes and the interactions between them. The technique used is based on CCS [14] and CSP [10].
- Abstract data types: this component describes the data types and value expressions. It is based on the abstract data type language ACT ONE [7].

5.1 Processes

A concurrent distributed system is described in LOTOS as a set of communicating processes. A process is like 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 actions and internal, external unobservable actions. Interactions between processes are achieved through synchronization. A synchronization is known as an *event*. An event is atomic and takes place at an *event gate* (or just *gate*).

As an example, let us consider the object model represented in Figure 1. As was said in the previous section, an object is specified as a process and one or more ADTs, where the process describes the dynamic behaviour of the object and the ADTs its state information. Suppose that the object `Account` offers the services `deposit` to credit an account and `get balance` to give the current balance of an account. The process could be specified as:

```
process Account[g](this_account: 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)
 ) >> accept updated_account: Account in Account[g](updated_account)
endproc
```

The process is defined recursively and uses gate `g` for synchronization with other processes. It communicates with other objects in the system by sending messages which are represented as events with the following structure:

<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;
```

The operator "!" is used in the form *!v* where *v* is a value expression. The operator "?" is used in the form *?v: s* where *v* is a variable of the sort *s*.

There are three kinds of synchronization which we summarise in Table 1.

Process A	**Process B**	**Condition**	**Interaction Type**
g ! E_1	g ! E_2	value (E_1) = value (E_2)	value matching
g ! E_1	g ? x: s	sort (E_1) = s	value passing
g ? y: w	g ? x: s	w = s	value generation

Table 1: Interaction Types

Value matching of `acc number` and `Get Account Number(this account)` is used to ensure correct synchronization. Although a client must know the identity of the server, a server can service many clients without knowing their identity. Value passing is used to pass the value `amount` to the variable `m`. Value generation allows the introduction of uninstantiated variables.

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 operator `accept ... in` is used to pass values as we exit from one process and enable another.

The functions `Get Account Number`, `Credit Account` and `Get Balance` are defined in the corresponding ADT. The parameter `this account` represents the object state information and is updated by the recursive call.

5.2 Abstract Data Types

LOTOS models data as abstract data types using the language ACT ONE. Their definition is rather lengthy and complex although this can be made easier by the provision of an extensive library of predefined ADTs.
The following example ADT defines the state of an account:

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

The list of imported definitions is given after the keyword **is**. The **sorts** section gives the name of the data sorts, the **opns** section defines the operations by their signature and the **eqns** section specifies, in terms of equations, the constraints the operations must satisfy. In section **eqns forall** we declare the variables that are going to be used in the equations and in section **ofsort** we define the result sort of the equations and then the equations themselves.

In ROOA we use *symbolic* ADTs. A symbolic 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 operation is performed internally.

6 Using the ROOA Method

In this section we show how to use ROOA, by using the automated banking system example given in Section 4.

Task 1: Build the Object Model

The object model produced by [18] is depicted in Figure 1.

Task 2: Refine the Object Model

As the object model only has attributes and static relationships, we have to normalize it by adding obvious services and message connections. To identify message connections, *interface scenarios* can be used. Interface scenarios model the interaction of a system with its environment. We can follow complete paths of functionality in the system, creating message connections as we trace the message passing through the object model. For example, as our system has to deal with accounts which can be credited, debited, etc., `deposit` and `withdraw` are events in the interface scenarios. We then have to guarantee that the system offers these services, by making them operations of the appropriate objects.

During this task we realized that some of the static relationships in the initial object model were in reality message connections. The normalized model is shown in Figure 2. The services are shown in the lowest third of each box, message connections are shown as arrows, and the two obvious hierarchies are marked by dotted lines. They correspond to the inheritance structures defined for tellers and accounts.

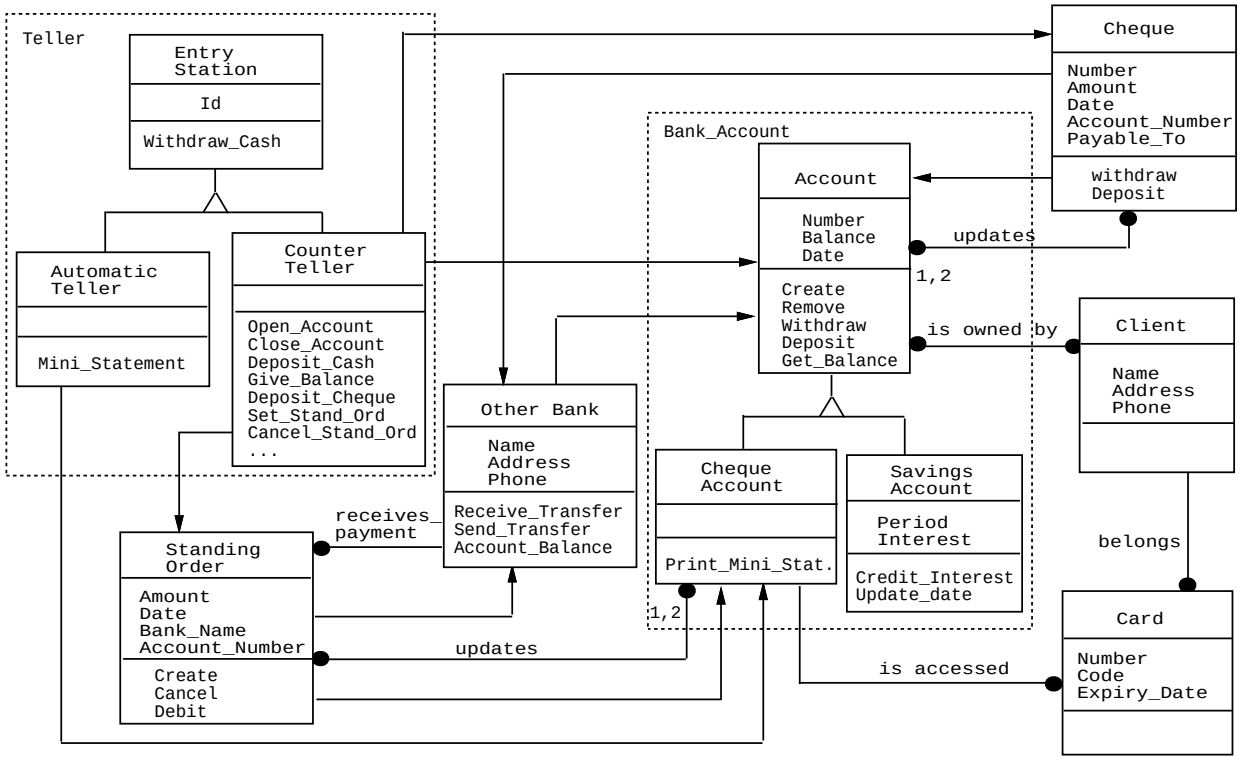

Figure 2: Normalized object model

Task 3: Build the LOTOS Formal Model

Task 3.1: Create an Object Communication Diagram

Each object in the object model is a node in the object communication diagram. The hierarchies identified in the previous task have to be shown.

`Teller` and `Other Bank` are the first clients in the system. `Cheque` and `Standing Order` embody the role of servers to `Teller` and of clients to `Bank Account`. `Bank Account` is the final server and so it can only communicate through gate `c` (see Figure 3).

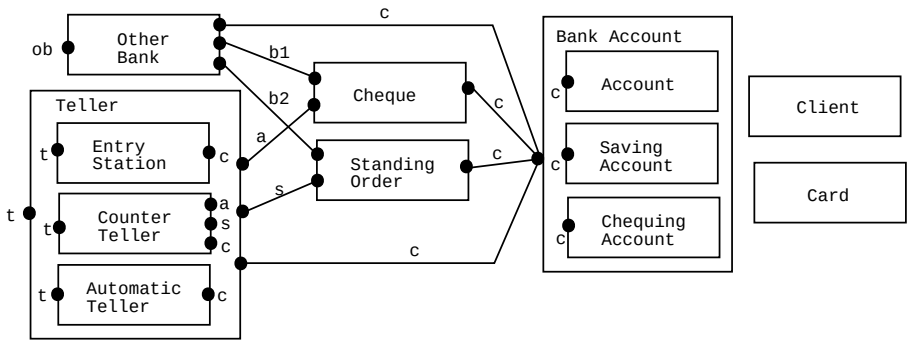

Figure 3: Initial object communication diagram

Notice that the objects `Card` and `Client` are not connected to the rest of the system. This will often be the case in a first iteration, but will be corrected as the method is applied. During later iterations new groupings will appear and the diagram will be modified.

Task 3.2: Specify Individual Objects and Classes

As our goal is to build a formal LOTOS specification, we have to specify objects and classes. An object definition can be specified in LOTOS as an ADT or as a process with one or more ADTs. We can start by specifying a process and its ADTs, by specifying a set of processes before dealing with ADTs, or start with the ADTs.

To specify the behaviour of an object we should place ourselves "inside" that object and act as if it was the centre of the system. By following this strategy we identify the events the object takes part in and their order. These events correspond to the operations the object offers to or requires of its environment and are often shown as the options of a choice expression.

As an example, let us look at the specification of the object `Account`. Figure 2 shows the services `Account` offers, each of which corresponds to an event. `Account` is, however, a superclass. Its subclasses, `Cheque Account` and `Savings Account`, inherit its properties and we can only specify inheritance in LOTOS if the superclass has **exit** functionality. We create a superclass where the operations offered by any account can be defined:

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

The definition of the account ADT is given in Section 5.2. Subclass `Cheque Account` can now be defined as:

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

Task 3.3: Compose the Objects into a Behaviour Expression

Once objects have been defined, they can be combined in a LOTOS behaviour expression which describes the whole or part of the system. We might, for example, initially ignore the `Cheque` and `Standing_Order` objects. The top-level behaviour expression would be:

```
 ((Teller[t, c] ||| Other_Bank[ob, c](bk of Bank_Name)) |[c]| Bank_Account[c])
|[t, ob]|
 Interface_Scenario[t, ob]

where
  process Bank_Account[c] : noexit :=
     Cheque_Account[c](acc2 of Account_Number)
   |||
     Savings_Account[c](acc1 of Account_Number)
  where ...
```

The interleaving operator `|||` indicates that `Teller` and `Other Bank` are composed in parallel, but do not interact with one another. The parallel operator `|[c]|` means that the behaviour expression

```
Teller[t, c] ||| Other_Bank[ob, c](bk of Bank_Name)
```

synchronizes in gate `c` with `Bank_Account[c]`.

It is often the case, as in this example, that we only require instances of the subclasses, not of their superclass. That is why `Account` does not appear in the behaviour expression for `Bank_Account`.

Task 3.4: Prototype the Specification

At this point we can start prototyping the specification, not only to identify syntactic and semantic errors, but also to validate the specification against the object model and the requirements. Interface scenarios are used to drive the prototyping.

Task 3.5: Refine the Specification

We have not yet dealt with static relationships. A relationship can be specified as an attribute, or a set of attributes, in one of the objects involved in the relation (or both if the relationship is bidirectional) [15]. Let us take the example of `Cheque Account`. As we can see from the object model, it has a relationship with `Card` and another with `Standing Order`. These two relationships are defined as ADTs given as parameters of the `Cheque Account` process. As the relationships are *one-to-many* in `Standing Order` and `Card` directions, they will be modelled as sets. This is shown by the parameters `cards` and `sos`.

```
process Cheque_Account[c](this_account: Account, cards: Card_Number_Set,
                          sos: SO_Number_Set) : noexit :=
 ( Superclass_Account[g](this_account)
    >> accept new_account: Account in exit(new_account, cards, sos)
 []
   c !print_mini_statement !Get_Account_Number(this_account) !this_account;
   exit(this_account, cards, sos)
 []
   c !perhaps_deposit !Get_Account_Number(this_account) ?m: Money;
   exit(Credit_Pending(this_account, m), cards, sos)
 []
   c !full_deposit !Get_Account_Number(this_account) ?m: Money ?valid: Bool;
   ( [valid] -> exit(Add_Credit_Pending(this_account, m), cards, sos)
    []
     [not (valid)] -> exit(Sub_Credit_Pending(this_account, m), cards, sos)
   )
 ) >> accept upd_account: Account, cards: Card_Number_Set, sos: SO_Number_Set
      in Cheque_Account[c](upd_account, cards, sos)
endproc
```

Note that the new version of `Cheque Account` extends both the state and the services that are inherited from `Superclass Account`. Also, that by specifying the relationships as parameters of the process, instead of specifying them in the `Account` ADT, we are promoting reusability.

During this task we have to introduce more detail in some of the processes in order to completely deal with the rest of the objects. That is why `perhaps deposit` and `full deposit` have been added. They are needed to deal with cheques. We also decided that `Card` and `Client` should only be specified as ADTs.

It is often the case that several instances of a class are required and we wish to be able to create the instances dynamically. This is the case with, for example, `Cheque Account` and is achieved by means of an *object generator* which is defined as:

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

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. As both kinds of account share the same `Account Number` sort, `!cheque` specifies the type of account we want to create and `Is Cheque Acc(acc counter)` guarantees that the new object identifier belongs to the correct subrange of `Account Number`.

The refinements lead us to the object communication diagram depicted in Figure 4.

The top-level behaviour expression is now:

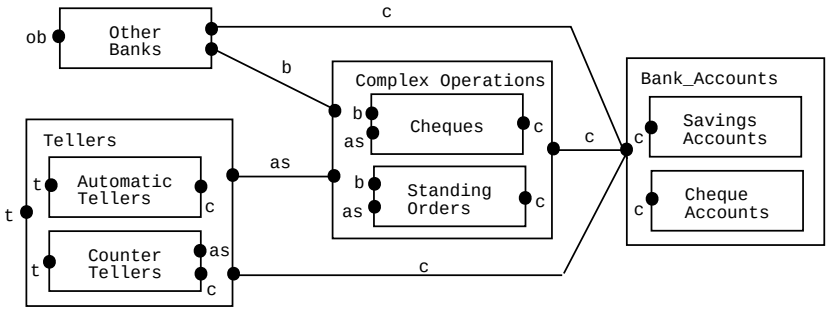

Figure 4: Final object communication diagram

```
(((Tellers[t, as, c] ||| Other_Banks[ob, b, c])
  |[as, b]|
   Complex_Operations[as, b, c]
 )
|[c]|
 Bank_Accounts[c]
) |[t, ob]| Interface_Scenario[t, ob]
```

7 Conclusions

The ROOA (Rigorous Object-Oriented Analysis) method integrates in a single formal model the object, dynamic and functional models usually proposed by the standard object-oriented analysis methods. 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 the clients who can check if the prototype exhibits the intended behaviour. Prototyping a formal specification enables omissions and inconsistencies in the original requirements to be readily identified. It also supports a software development trajectory where the requirements specification is transformed into a design specification with prototyping being used to ensure that the two specifications conform to one another.

The dynamic behaviour of each object is specified as a LOTOS process and its state information can be specified by one or more ADTs. The processes are composed, by using the LOTOS parallel operators, to specify the dynamic behaviour of the complete system. Therefore we can specify a system as a set of concurrent objects and avoid decisions that can be considered design or implementation issues, such as protection techniques for the concurrent access of shared data. Much of the concurrency will be removed in an implementation, but we are performing analysis, and therefore our goal is to understand the problem, not to propose a solution.

References

[1] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[2] E. Brinksma (ed). *Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique on the Temporal Ordering of Observation Behaviour, ISO 8807*, 1988.

[3] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An Object-Oriented Extension to Z. In Son T. Vuong, editor, *Formal Description Techniques, II*, pages 281–295, North-Holland, December 1989.

[4] R.G. Clark. Using LOTOS in the Object-Based Development of Embedded Systems. In C.M.I. Rattray and R.G. Clark, editors, *Unified Computation Laboratory*, pages 307–319. Oxford University Press, 1992.

[5] P. Coad and E. Yourdon. *Object Oriented Analysis*. Yourdon Press, Prentice-Hall, 2nd edition, 1991.

[6] H. Eertink. Executing LOTOS Specifications: The SMILE Tool. In *Third LotoSphere Workshop and Seminar*, September 1992.

[7] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications*, volume 1. Springer-Verlag, 1985.

[8] N.E. Fuchs. Specifications are (preferably) Executable. *Software Engineering Journal*, 7(5):323–334, 1992.

[9] I.J. Hayes and C.B. Jones. Specifications are not (Necessarily) Executable. *Software Engineering Journal*, 4(6):330–338, November 1989.

[10] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[11] ISO/IEC JTC1/SC21/WG7. Basic Reference Model of Open Distributed Processing. Technical report, 1993.

[12] I. Jacobson. *Object-Oriented Software Engineering — A Use Case Driven Approach*. Addison-Wesley, 1992.

[13] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1986.

[14] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[15] A.M.D. Moreira and R.G. Clark. Rigorous Object-Oriented Analysis. Technical Report TR 109, Computing Science Department, University of Stirling, Scotland, 1993.

[16] K.S. Rubin and A. Goldberg. Object Behaviour Analysis. *Communications of the ACM*, 35(9):48–62, 1992.

[17] S. Rudkin. Inheritance in LOTOS. In K.R. Parker and G.A. Rose, editors, *Formal Description Techniques, IV*, pages 409–423, North-Holland, 1992.

[18] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall, 1991.

[19] S. Shlaer and S.J. Mellor. *Object Lifecycles — Modeling the World in States*. Prentice-Hall, 1992.