Combining Object-Oriented Analysis and Formal Description Techniques

Ana M. D. Moreira and Robert G. Clark

Department of Computing Science and Mathematics
University Of Stirling
STIRLING FK9 4LA, Scotland, UK
amm@cs.stir.ac.uk
rgc@cs.stir.ac.uk

Abstract. The Rigorous Object-Oriented Analysis (ROOA) method provides a systematic development process by proposing a set of rules to be followed during the analysis phase. ROOA takes 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 other object-oriented analysis methods which are informal and produce three separate models that are difficult to integrate and keep consistent. The model is expressed in LOTOS and provides a precise and unambiguous specification of a system's requirements. As the specification obtained is executable, 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.

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 [17] we describe the ROOA (Rigorous Object-Oriented Analysis) method in detail[1]. In this paper, we give an example of the application of the method. The purpose of this is to demonstrate that the method is useful in practical situations. ROOA integrates the static properties captured in an object model produced by any object-oriented analysis method [5, 12, 19, 21] with the dynamic and functional properties given in the original requirements and produces an executable formal object-oriented model that acts as a requirements specification. Formal description techniques, such as [1, 3, 13], are usually applied

[1] A shorter description of the ROOA method is presented in Portuguese in [16].

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 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 an introduction to object-oriented analysis methods. Section 4 summarizes the ROOA method. Section 5 presents the problem we use to illustrate LOTOS and the ROOA method. Section 6 introduces LOTOS. Section 7 shows how the ROOA method can be used to derive a formal LOTOS object-oriented analysis model.

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 useful. 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 allows the use of uninstantiated variables within conditions and is able to determine when a combination of conditions can never be true [6]. 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 (OOA) 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 classes.
4. Identify relationships between objects.
5. Identify inheritance relationships between classes.
6. 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 objects, several methods [5, 21] suggest we look at nouns, pronouns, noun phrases, adjectival and adverbial phrases in the initial requirements document, while others [19] suggest that a better way of identifying objects is to focus on their behaviour. Once objects have been identified, they are grouped 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 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 add *scenarios* [21] (or *use cases* [12]) to the dynamic model and show the interactions between objects for each scenario by means of an *event trace diagram* [21] (or *interaction diagram* [12]).

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 [21, 22], 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.

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 Eiffel [14] 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 by using any of the existing OOA methods. In the second task we refine the object model to ensure that it incorporates interface objects, attributes, services, static relationships and message connections, and we identify object groupings. In the third task we build the formal LOTOS object-oriented analysis model. The ROOA method gives a formal interpretation in LOTOS of object-oriented analysis constructs.

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. Figure 1 shows the composition of the two main tasks of ROOA, *Refine Object Model* and *Build Formal LOTOS OOA Model.*

Task 1: Build the Object Model. Before we start producing the formal model, we have to build an object model by using any existing OOA method [5, 12, 19, 21]. 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. An advantage of starting with an object model produced by any OOA method is that we build on the work which has already been done to identify objects.

During the application of the ROOA method, the requirements document and the object model may be modified.

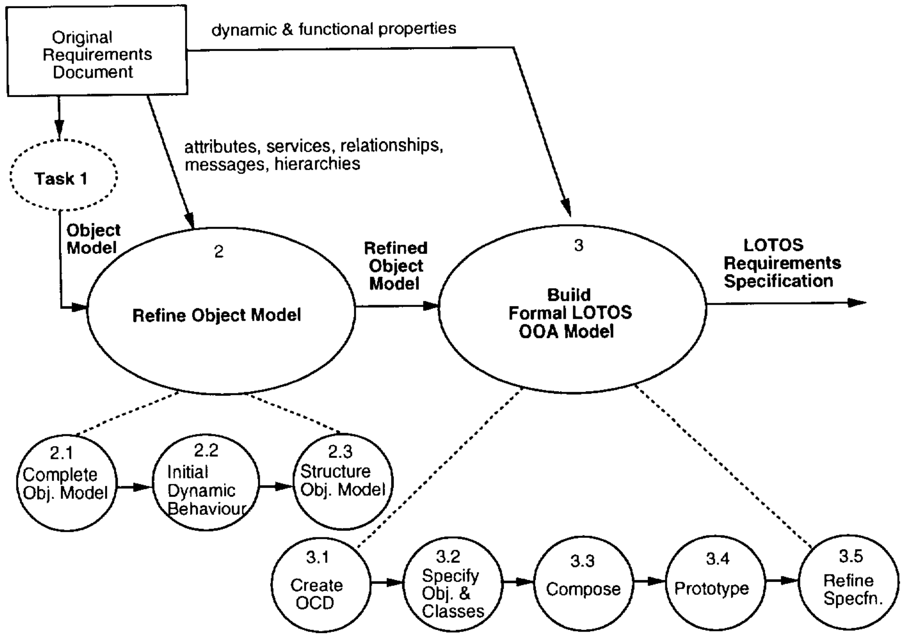
Original Requirements Document
dynamic & functional properties
attributes, services, relationships, messages, hierarchies
Task 1
Object Model
2
Refine Object Model
Refined Object Model
3
Build Formal LOTOS OOA Model
LOTOS Requirements Specification
2.1 Complete Obj. Model
2.2 Initial Dynamic Behaviour
2.3 Structure Obj. Model
3.1 Create OCD
3.2 Specify Obj. & Classes
3.3 Compose
3.4 Prototype
3.5 Refine Specfn.

Fig. 1. Core of ROOA

Task 2: Refine the Object Model. Before we start producing the LOTOS formal model, we must ensure that our object model is complete by adding, if they are not already present, static relationships, attributes, services, message connections and interface objects.

An interface object models behaviour and information that is part of the system interface with the system's environment. We also define interface scenarios. An interface scenario shows a series of services (requests and responses) that the actors (clients or users) can require from the system. It can be seen as a list of calls to the services offered by the interface objects together with the expected responses. Its effect within the system is described by an event trace diagram.

In Task 2.2, we start building the Object Communication Table (OCT) which will be completed in Task 3.1. Eventually, this table will be composed of five columns, but now we only build the first four columns. In the first column (*Objects*) we list the objects that form the object model; in the second column (*Offered Services*) we list the services offered by each object; in the third column (*Required Services*) we list, for each service offered in column two, the services that it requires from other objects; and, in the fourth column (*Clients*) we list, for each offered service, the objects (clients) which require that service. We use the notation <*object.service*> to indicate that the *service* defined in *object* is required. Event trace diagrams are used in constructing columns two, three and four of the OCT. Part of an OCT is shown in Table 2.

In Task 2.3, we structure the object model by identifying groupings of objects in order to make the system easier to understand and develop. As this is difficult, we cannot expect to do it completely and correctly in the first iteration and so we return to this task when our knowledge about individual objects has increased. Candidates for grouping are aggregates, a superclass and its subclasses, a set of clients which use the same servers, and a set of servers which have the same clients. The low level objects in the object model often remain almost unchanged during the development, but the high level structure is less stable.

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.

Create an Object Communication Diagram (OCD). This diagram is a graph in which, 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 generalized 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.

Before we start building the OCD, we complete the OCT by adding the column *Gates* which gives the name of the gates that the objects in column one and column four use to communicate with each other. This information is then used to label the arcs in the OCD.

We follow three basic rules to define gates of communication between a server and its clients:

1. Use the same gate for object communications which require the same set, or subset, of services; e.g. where there is an overlap between the set of services required by different clients, from a given server, we use a single gate for communication.
2. Use different gates for object communications which require a different set of services; e.g. where there is no overlap between the set of services required by each client from a server, we give different gates of communication for each client involved in the study.
3. Two objects at the same level of abstraction which communicate through a given gate cannot use this gate to communicate with other objects at a different level of abstraction.

Specify individual objects and classes. In [17] we show how to model object-oriented constructs in LOTOS, in particular, objects, class templates and classes. A class template describes the common static and dynamic properties of objects of the same kind (belonging to the same class). A class is the set of all objects which share the common features specified by a class template. An object is a member of a class and is created by instantiating a class template.

The behaviour of an object is specified by a class template and its state information by one or more Abstract Data Types (ADTs) given as parameters of the template. And so, for each individual object, we:

1. Specify the class template by a LOTOS process definition, identifying the events it takes part in and their order.
2. Specify ADTs to describe its attributes.

Inheritance in LOTOS is more of a problem and a theoretical study has been made by Rudkin [20]. 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 and/or modifies.

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 an example is given in Sect. 7. In [18], we describe how incremental inheritance in LOTOS can be restricted so that behavioural inheritance is guaranteed.

Compose the objects into a behaviour expression. Following the structure of the object communication diagram, we compose the objects into a LOTOS behaviour expression by using the LOTOS parallel operators. We have an algorithm which converts an OCD into a LOTOS behaviour expression and identifies the few situations in which an OCD cannot be represented in LOTOS [17]. The important rule is that if a server has several clients at the same gate then the server can either be grouped with all the clients or with none of them.

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 requirements, the object model, the OCT, the OCD and the specification.

Refine the specification. The specification is refined by re-applying the whole or part of the process. During successive refinements we may identify new higher level objects, model static relationships, define object generators so that multiple instances of the same class template 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.

5 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 services 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 chequing accounts. Savings accounts give interest and cannot be accessed by the automatic tellers.

We applied the object-oriented analysis methods of OOA [5] and OMT [21] to this problem, but only the object model produced by OMT is presented here (see Fig. 2). The nodes in the object model represent class templates with attributes. Relationships between objects are represented by a line connecting two class templates. A relationship has a name and cardinality. Cardinality of exactly one is shown as a straight line while zero-or-more is shown with an added filled circle. An inheritance hierarchy is represented by a triangle.

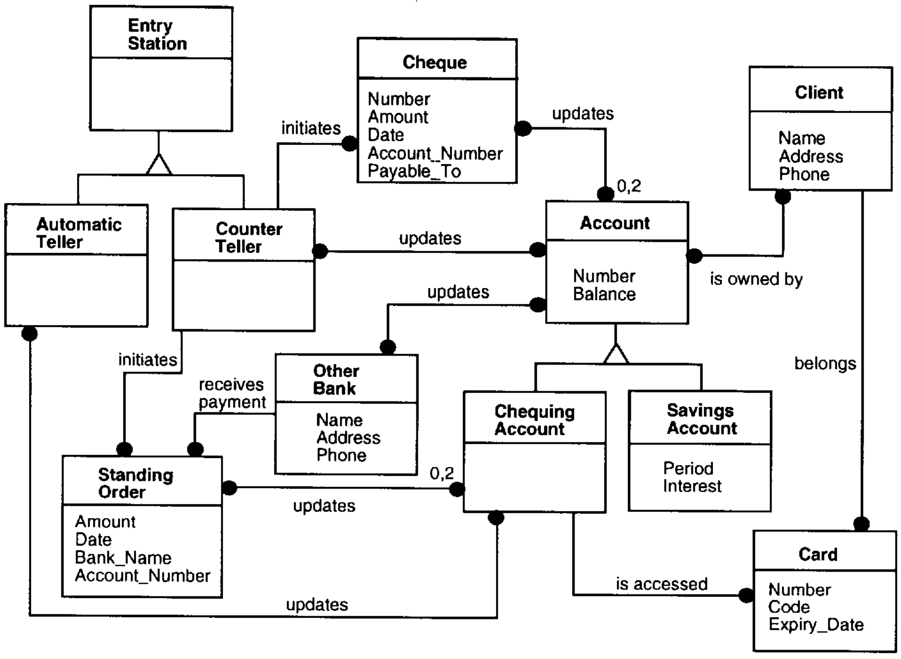

Fig. 2. Object model produced by the OMT method

6 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 approach used is based on process algebra, using components from CCS [15] 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].

6.1 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 events and internal, externally unobservable events. Interactions between process instantiations are achieved through synchronization. A synchronization is known as an *event*. An event is atomic and takes place at a *gate*.

As an example, let us consider the object model represented in Fig. 2. As was said in the previous section, a class template is specified as a process and one or more ADTs, where the process describes the dynamic behaviour of the class template and the ADTs its state information. Suppose that the class template `Account` offers the services `deposit` to credit an account, `withdraw` to debit 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)
  []
    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 new_account: Account in Account[g](new_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> <message name> <object identifier> <optional parameters>

Each service in the object model has a corresponding message name in a LOTOS structured event and each event may cause the execution of ADT operations. The service and the message have the same name. 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 summarize in Table 1 [1].

Table 1. Interaction types

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

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 the generalized choice operator **choice**, used to specify the service `withdraw`, allows the specification of the two possible situations with an account (the account has funds and the account has no funds). The >> 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**, **Debit_Account** and **Get_Balance** are defined as operations in the **Account** ADT. The parameter **this_account** represents the object state information and is updated by the recursive invocation of process **Account**.

6.2 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.

In ROOA we define 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. This helps reduce the length of an ADT, saving time during the construction of the specification. More detail will be added in the design phase.

ADT operations can be classified as constructors, modifiers or selectors. In ROOA, an ADT is built in the following way:

- *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.
- *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 of the operation. 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 all the different possible situations are covered.
- *Define equations for selectors that need to return a particular value.* The selector must be defined using an equation for each constructor.

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 Make_Account : Account_Number, Balance -> Account
       Credit_Account : Account, Money -> Account
       Debit_Account : Account, Money -> Account
       Get_Balance : Account -> Balance
       Get_Account_Number : Account -> Account_Number
       ...
  eqns forall a: 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
```

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 `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 a 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, the generalized **choice** operator in process `Account` enables us to explore the two possible situations: either there is enough money in an account or there is not enough money. `Get_Account_Number`, however, has to return a particular account number and so it is defined with an equation for each constructor.

7 Using the ROOA Method

In ROOA, we use LOTOS to specify the required behaviour of a proposed system by building a formal model, i.e. a model which is expressed in a language which has a formal semantics. The model describes the required behaviour in terms of a set of communicating concurrent objects, where each object is represented as the instantiation of a LOTOS process and the communication between two objects is represented by the two processes synchronizing on an event. As LOTOS has a formal semantics, the model of the required behaviour has a precise meaning and can, therefore, be used as a formal requirements specification of the intended system behaviour.

In this section we show how to use ROOA, by using the automated banking system example given in Sect. 5.

Task 1: Build the Object Model.

The object model produced by [21] is depicted in Fig. 2.

Task 2: Refine the Object Model.

As the object model only has attributes and static relationships, we have to complete it by adding services and message connections. In order to accomplish this, we use interface scenarios together with event trace diagrams. 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 or debited, `deposit` and `withdraw` are events in the interface scenarios.

During this study we start building the OCT. Table 2 shows part of the initial OCT for the objects `Counter_Teller`, `Other_Bank` and `Account`. (`SO` stands for `Standing_Order` and `ES` stands for `Entry_Station`.)

Table 2. Initial OCT

Objects	Services Offered	Services Required	Clients
Counter_Teller (CT)	open_account deposit_cash give_balance ask_transfer ...	Account.create Account.deposit Account.get_balance Account.withdraw Account.deposit OB.send_transfer	Interface_Scenario Interface_Scenario Interface_Scenario Interface_Scenario
Other_Bank(OB)	receive_transfer send_transfer cheque_withdraw remote_withdraw	Account.deposit Account.withdraw	Interface_Scenario CT, SO Cheque Interface_Scenario
Account(A)	create deposit withdraw get_balance ...		CT CT, Cheque, OB, SO CT, ES, Cheque, OB, SO CT

During this task we realized that some of the static relationships in the initial object model were in reality message connections. The refined model is shown in Fig. 3. The services are shown in the lowest third of each box, message connections are shown as arrows, and the two obvious groupings of objects are marked by dotted lines. They correspond to the inheritance structures defined for tellers and accounts.

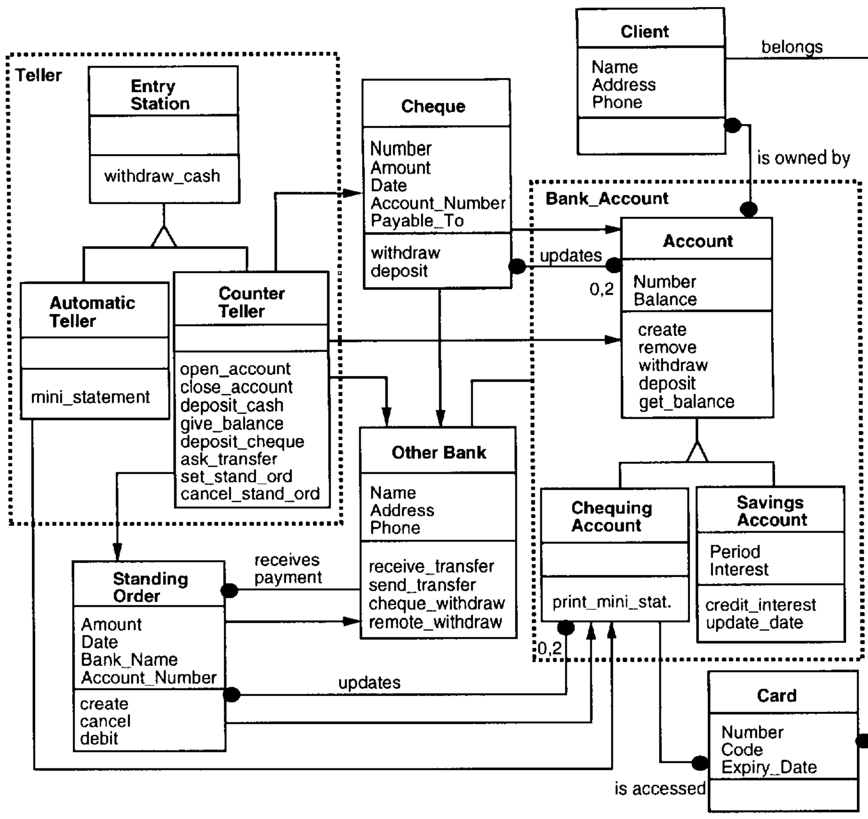

Fig. 3. Refined object model

Task 3: Build the LOTOS Formal Model.

Task 3.1: Create an Object Communication Diagram. We now add gates to the OCT (see Table 3) and show the obvious groupings: `Teller` composed of `Entry_Station`, `Counter_Teller` and `Automatic_Teller` (`AT`); `Bank_Account` composed of `Account`, `Chequing_Account` (`CA`) and `Savings_Account` (`SA`).

The OCD is built directly from the OCT. Each node in the OCD is an entry in column one of the OCT, the arcs are given by analysing columns one and four and named by column five. The hierarchies identified in the previous task are 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 `ba`. Figure 4 shows the initial OCD.

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.

Table 3. OCT with gates

Objects	Services Offered	Services Required	Clients	Gates
Teller	open_account(CT)	BA.create	Interface_Scenario	t
[ES + CT + AT]	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
	ask_transfer(CT)	BA.withdraw	Interface_Scenario	t
		BA.deposit		
		OB.send_transfer		
	...			
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
Bank_Account(BA)	create(A)		Teller(CT)	ba
[A + CA + SA]	deposit(A)		Teller(CT), OB, Cheque, SO	ba
	withdraw(A)		Teller(ES,CT), OB, Cheque, SO	ba
	get_balance(A)		Teller(CT)	ba
	...			

Task 3.2: Specify Individual Objects and Classes. As our goal is to build a formal LOTOS specification, we have to specify objects, class templates and classes. If an object only plays a minor role in the system, it can be modelled as an attribute of a more important object in which case it is specified as an ADT. Otherwise, an object is specified 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 we were 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 services 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 `Chequing_Account` which inherits from the `Account` superclass. In Sect. 6, we presented a definition of an `Account` process and of an `Account` ADT. The ADT remains unchanged, but if `Account` is to be a superclass then the process definition must be modified.

The earlier version of `Account` has **noexit** functionality. After a service has been handled, `Account` is invoked recursively. The new version of `Account` has **exit** functionality. The reason for the change is that, within the specification of the `Chequing_Account` subclass, many of the offered services are provided by

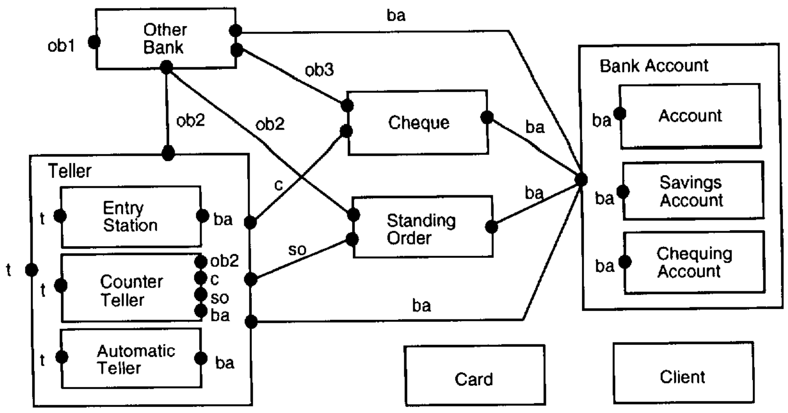

Fig. 4. Initial object communication diagram

invoking `Account`. After a service defined in `Account` has been handled, all the services offered by `Chequing_Account` must again be on offer. We must therefore exit from process `Account` so that `Chequing_Account`, and not `Account`, is invoked recursively. We first define the `Account` superclass:

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

We now define the subclass `Chequing_Account` by extending `Account` with the additional service `print_mini_statement`.

```
process Chequing_Account[ba](this_account: Account): noexit :=
  ( Account[ba](this_account)
  []
    ba !print_mini_statement !Get_Account_Number(this_account)
       !this_account;
    exit(this_account)
  ) >> accept updated_account: Account
       in Chequing_Account[ba](updated_account)
endproc
```

After a service has been handled, `Chequing_Account` is invoked recursively and so all the services are again on offer.

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 interactions with the Cheque and Standing_Order objects. The top-level behaviour expression would be:

```
( ( Other_Bank[ob1, ob2, ba](Make_Bank(...))
    |[ob2]| Teller[t, ob2, ba]
  )
  |[ba]| Bank_Account[ba]
)
|[t, ob1]| Interface_Scenario[t, ob1]

where
process Bank_Account[ba]: noexit :=
  Chequing_Account[ba](Make_Account(acc1 of Account_Number, 0))
  |||
  Savings_Account[ba](Make_Account(acc2 of Account_Number, 0))
where ...
```

The interleaving operator ||| indicates that the objects Chequing_Account and Savings_Account are composed in parallel, but do not interact with one another. The parallel operator |[ob2]| means that the objects Other_Bank and Teller communicate with each other on gate ob2.

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

Task 3.4: Prototype the Specification. Tools are available to identify syntax and semantic errors and to prototype the LOTOS specification. Interface scenarios are used to drive the prototyping so that the specification can be validated against the object model and the requirements. The SMILE simulator [6] allows the use of uninstantiated variables within conditions and uses a *narrowing algorithm* 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.

Task 3.5: Refine the Specification. We have not yet dealt with static relationships. A relationship is modelled as an argument in the process defining the class template. This argument is an ADT which represents either the identifier of an object or a set of identifiers, depending on the cardinality of the relationship [17].

Let us take the example of Chequing_Account. As we can see from the object model, it has a relationship with Card and another with Standing_Order. These relationships are defined as ADTs given as parameters of the Chequing_Account process. As the relationships have cardinality *many* in Standing_Order and Card directions, they will be modelled as sets. This is shown by the parameters cards and sos.

```
process Chequing_Account[ba](this_account: Account,
        cards: Card_Number_Set, sos: SO_Number_Set): noexit :=
  ( Account[ba](this_account) >> accept new_account: Account
       in exit(new_account, cards, sos)
  []
    ba !print_mini_statement !Get_Account_Number(this_account)
       !this_account;
    exit(this_account, cards, sos)
  []
    ba !perhaps_deposit !Get_Account_Number(this_account)
       ?m: Money;
    exit(Credit_Pending(this_account, m), cards, sos)
  []
    ba !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 updated_account: Account, cards: Card_Number_Set,
              sos: SO_Number_Set
       in Chequing_Account[ba](updated_account, cards, sos)
endproc
```

Note that the new version of Chequing_Account extends both the state and the services that are inherited from Account. Specifying relationships as parameters of the process, instead of specifying them in the Account ADT, promotes reusability of the ADTs.

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

In general, several instances of a class are required and we wish to be able to create the instances dynamically. This is achieved by means of an *object generator*. In the case, for example, of Chequing_Account this is defined as:

```
process Chequing_Accounts[ba](accs: Account_Number_Set): noexit :=
  ba !create !cheque ?acc_counter: Account_Number
     [(acc_counter notin accs) and Is_Chequing_Acc(acc_counter)];
  ( Chequing_Account[ba](Make_Account(acc_counter, 0),
                        {} of Card_Number_Set, {} of SO_Number_Set)
    |||
    Chequing_Accounts[ba](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_Chequing_Acc(acc_counter)]
```

imposes the condition that the new object identifier differs 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_Chequing_Acc(acc_counter)` guarantees that the new object identifier belongs to the correct subrange of `Account_Number`.

During this task we have also grouped `Cheques` with `Standing_Orders` to form the subsystem `Complex_Operations`. The OCT has to be changed to reflect this grouping and the rules to name gates have to be applied again. The refinements lead us to the object communication diagram depicted in Fig. 5.

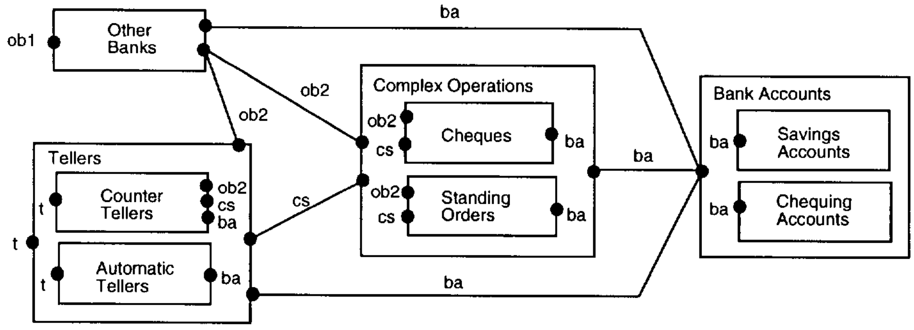

Fig. 5. Final object communication diagram

The top-level behaviour expression is now:

```
( ( Other_Banks[ob1, ob2, ba]
        (Insert(This_Bank, {} of Bank_Name_Set))
    |[ob2]|
    ( Tellers[t, ob2, cs, ba]
      |[cs]| Complex_Operations[ob2, cs, ba]
    )
  )
  |[ba]| Bank_Accounts[ba]
)
|[t, ob1]| Interface_Scenario[t, ob1]
```

The refinement of the formal model is both incremental and iterative. As information is added, more static relationships, attributes, services and message connections can be identified. At all stages we must ensure that the model is internally consistent and the final model must deal with all the essential objects identified in the original requirements.

8 Conclusions

The ROOA (Rigorous Object-Oriented Analysis) method consists of three main tasks: building an object model, refining the object model and building a formal LOTOS OOA model. The resulting formal model integrates the object, dynamic and functional models usually proposed by object-oriented analysis methods. The development process is iterative and parts of the method can be re-applied to subsystems.

The dynamic behaviour of each object is specified by 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 in the analysis phase, and therefore our goal is to understand the problem, not to propose a solution.

As LOTOS has a precise mathematical semantics, the LOTOS 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.

ROOA also provides the first stage in a software development trajectory where a requirements specification is transformed into a design specification with prototyping being used to ensure that the two specifications conform to one another.

9 Acknowledgments

This work has been realized with the support of Junta Nacional de Investigação Científica e Tecnológica (JNICT), Portugal.

References

1. Bolognesi, T., Brinksma, E.: Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14, 25-59, 1987.
2. Brinksma E. (ed).: *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, ISO 8807, 1988.
3. Carrington, D., Duke, D., Duke, R., King, P., Rose, G., Smith, G.: Object-Z: An Object-Oriented Extension to Z. In: Vuong S.T. (ed): *Formal Description Techniques II*, North-Holland 1989, pp. 281-295.
4. Clark, R.G.: Using LOTOS in the Object-Based Development of Embedded Systems. In: Rattray C.M.I., Clark R.G. (eds): *Unified Computation Laboratory*, Oxford University Press 1992, pp. 307-319.

5. Coad, P., Yourdon, E.: *Object Oriented Analysis (Second Edition)*, Yourdon Press, Prentice-Hall 1991.
6. Eertink H., Wolz D.: Symbolic Execution of LOTOS Specifications. In: Diaz M., Groz R. (eds): *Formal Description Techniques V*, North-Holland 1993, pp. 295-310.
7. Ehrig, H., Mahr, B.: *Fundamentals of Algebraic Specifications* (Vol. 1), Springer-Verlag 1985.
8. Fuchs, N.E.: Specifications are (preferably) Executable. *Software Engineering Journal*, **7**, 323-334, 1992.
9. Hayes, I.J., Jones, C.B.: Specifications are not (Necessarily) Executable. *Software Engineering Journal*, **4**, 330-338, 1989.
10. Hoare, C.A.R.: *Communicating Sequential Processes.* Prentice Hall 1985.
11. ISO/IEC JTC1/SC21/WG7: *Basic Reference Model of Open Distributed Processing*, 1993.
12. Jacobson, I.: *Object-Oriented Software Engineering.* Addison-Wesley 1992.
13. Jones, C.B.: *Systematic Software Development Using VDM.* Prentice Hall 1986.
14. Meyer, B.: *Eiffel: The Language.* Prentice Hall 1992.
15. Milner, R.: *Communication and Concurrency.* Prentice-Hall 1989.
16. Moreira, A.M.D., Clark, R.G.: Os Métodos Formais na Análise de Orientação por Objectos. In: Leite, J. (ed): *Proceedings 7th Brazilian Symposium on Software Engineering*, Rio de Janeiro, October 1993, pp. 238-252.
17. Moreira, A.M.D., Clark, R.G.: Rigorous Object-Oriented Analysis. Technical Report TR 109, Computing Science Department, University of Stirling, Scotland 1993.
18. Moreira, A.M.D., Clark, R.G.: LOTOS in the Object-Oriented Analysis Process. In: *BCS-FACS Workshop on Formal Aspects of Object Oriented Systems*, London, December 1993.
19. Rubin, K.S., Goldberg, A.: Object Behaviour Analysis. *Communications of the ACM*, **35**(9) 48-62, 1992.
20. Rudkin, S.: Inheritance in LOTOS. In: Parker, K.R., Rose, G.A. (eds): *Formal Description Techniques IV*, North-Holland 1992, pp. 409-423.
21. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: *Object-Oriented Modelling and Design*, Prentice-Hall 1991.
22. Shlaer, S., Mellor, S.J.: *Object Lifecycles — Modeling the World in States*, Prentice-Hall 1992.