ROOA:
Rigorous Object-Oriented Analysis Method

Ana M. D. Moreira
and
Robert G. Clark

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

Abstract

Object-oriented analysis (OOA) and design methods are used by the software engineering community, while formal description techniques (FDTs) are mainly used in a research environment. The Rigorous Object-Oriented Analysis (ROOA) method combines OOA methods with the ISO standard FDT LOTOS to produce a practical method which can be applied by software engineers.

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 existing OOA methods which are informal and produce three separate models that are difficult to integrate and keep consistent.

The ROOA method 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.

An important part of the ROOA method is to give a formal interpretation in LOTOS of object-oriented analysis constructs.

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.

LOTOS is a wide-spectrum specification language and so the requirements specification can act as the starting point for software development based on correctness preserving transformations.

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. In the Rigorous Object-Oriented Analysis (ROOA) method, described in this document, they are applied in combination to the requirements analysis phase of the software life cycle.

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 produced, 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. We have been investigating how these tasks can be achieved within the context of object-oriented analysis.

In the ROOA method, we specify the required behaviour of a system by constructing a model using the formal description technique LOTOS [4, 5]. As LOTOS has a formal semantics, the model has a precise meaning and can be used as a formal requirements specification of the intended system behaviour.

Object-oriented analysis (OOA) methods such as [9, 15, 24, 25] lead to the creation of an informal analysis model or set of models. The object model, which is based on an extension to entity-relationship diagrams, describes the static properties of a system while the dynamic model, which is normally expressed in terms of state transition diagrams, describes its behaviour. Some methods [24, 25] 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. 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 formalize the analysis model should therefore support parallelism.

ROOA shows how LOTOS can be integrated with object-oriented analysis. LOTOS is composed of a process algebra and abstract data types, but, as it was designed before the object-oriented approach became widely accepted, it does not directly incorporate object-oriented constructs. However, the language is suitable for writing specifications in an object-oriented style [22].

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

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. The ROOA method builds on the object model produced by one of the informal OOA methods. However, instead of producing three separate models, the ROOA method uses a set of rules to systematically create a single integrated object-oriented analysis model from the static properties captured in the object model and the dynamic and functional properties given in the informal requirements. The model is formal as it is expressed in a language which has a formal semantics. During the application of the rules, auxiliary structures are created to help trace the requirements through to the final formal model and to show when and why information from the informal requirements is added to the formal model. This is important when modifications are made to the system requirements.

An important part of the ROOA method is to give 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.

As a LOTOS specification is executable, prototyping may be used to discover omissions, contradictions, ambiguities or inconsistencies early in the development process and to demonstrate to the clients that the specification meets their requirements. A set of tools (such as syntax and semantic checkers, simulators) is available with LOTOS. The SMILE simulator [11] supports value generation and allows symbolic execution of a specification where a set of possible values is used rather than particular values. In the analysis phase, nondeterminism can be exploited to model behaviour so that premature design decisions are not made.

Chapter 2 introduces the formal description technique LOTOS. Chapter 3 presents the most common object-oriented analysis concepts and discusses how to specify them in LOTOS. Chapter 4 focuses on the ROOA method, detailing each task and describing the intermediate structures that we need to build in order to develop the final object-oriented analysis LOTOS model and to help tracing through the original requirements to the LOTOS specification. Chapter 5 states the conclusions of this work and discusses further development.

Chapter 2

LOTOS Overview

2.1 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 [6]. 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 [19] and CSP [13] .
- Abstract data types.
  This component describes the data types and value expressions. It is based on the abstract data type language ACT ONE [12].

2.2 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 2.1 summarizes the three types of synchronization [4].

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 2.1: 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.

2.3 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.

2.4 Overall Structure

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 2.2 summarizes the syntax of the most common behaviour expressions. These can be combined to define complex behaviour expressions.

Name	**Syntax**
inaction	**stop**
termination	**exit**
termination with parameters	**exit**$(E_1,\ldots,E_n)$
choice	B_1 [] B_2
generalized-choice	**choice** $v{:}T$ [] B
	choice g **in** $[g_1, \ldots, g_n]$ [] 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 |\[$g_1, \ldots, g_n$\]| B_2
interleaving	B_1 ||| B_2
full synchronization	B_1 || B_2
hiding	**hide** $g_1,\ldots,g_n$ **in** B
process instantiation	$P[g_1,\ldots,g_n]\ (E_1\ldots E_n)$
guarding	$[CE]$ -> B
disabling	B_1 [> B_2
enabling	B_1 >> B_2
enabling with variable passing	B_1 >> **accept** $v_1{:}\ T_1,\ldots,v_n{:}\ T_n$ **in** B_2
local definition	**let** $v_1{:}\ T_1 = E_1,\ldots,v_n{:}\ T_n = E_n$ **in** B

Legend:

B, B_1, B_2 : behaviour expressions	$T_1,\ldots, T_n$: sort identifiers
$v_1,\ldots, v_n$: variable identifiers	$E_1,\ldots, E_n$: value expressions
$g_1,\ldots, g_n$: gate identifiers	CE : conditional expression
$d_1,\ldots, d_n$: experiment offers	P : process identifier

Table 2.2: Syntax of the most important LOTOS operators

Chapter 3

Modelling Basic Object-Oriented Concepts in LOTOS

3.1 Introduction

LOTOS was designed before object-oriented methods became established and therefore it does not incorporate object-oriented constructs. However, the language directly supports the concepts of encapsulation, abstraction and information hiding which provide a basis for writing specifications in an object-oriented style [3, 8, 10, 16, 22]. In LOTOS, concurrent objects are modelled as process instances, composed by using the parallel operators, and message passing is modelled as two processes synchronizing on an event.

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

- Class templates, classes, abstract classes and objects.
  - Services, methods and attributes.
  - Object identity.
  - State, behaviour and environment of an object.
- Message connections (communication).
  - Events.
  - Transformation of data and information passed.
- Inheritance.
  - Superclasses and subclasses.
  - Extension and redefinition.
- Associations.
  - Binary and unary.
  - With values.
- Composition and Decomposition.

- Aggregation.
- Subsystems.

In Section 3.2 we describe each of these object-oriented concepts and in Section 3.3 we give an interpretation of them in LOTOS.

3.2 Object-Oriented Definitions

The following definitions are based on the Open Distributed Processing Reference Model [14] and on a general understanding of object-oriented analysis concepts.

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 box 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 Classes

An abstract class is a class which cannot be instantiated, i.e. it cannot have objects. This restriction does not apply to its descendents. An abstract class is used in the definition of subclasses.

Some authors, such as [9], propose a special abstract class symbol to be used in the object model.

3.2.4 Objects

An *object* is a model of either a real world entity or a concept. It combines structure with behaviour in a single encapsulated entity which can be characterized by *name*, *state information* and *services* (or *operations*). 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. Only in the design and implementation phases is its internal view studied.

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.

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

3.2.5 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 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 initializes 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 cannot change the state.

3.2.6 Methods

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

3.2.7 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.

3.2.8 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. In [24] identity is defined as:

> 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*.

3.2.9 State (of an Object)

The state of an object, at a given time, is given by the values of its attributes and by the conditions that determine the events in which the object can take part. The values of the attributes are related to the static aspect of the object, while the conditions control the possible state transitions and therefore are related to the dynamic aspect of the object.

3.2.10 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 that object changes.

3.2.11 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.

3.2.12 External Objects

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

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, the order of two events may, or may not, be related. One event may logically 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 the object model as arrows. If a message connection is defined from object A to object B, it means that object A requires operations 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*, equivalent to remote procedure calls.

3.2.15 Inheritance

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

There are two main definitions of inheritance [14]: *behavioural inheritance* and *incremental inheritance*. Behavioural inheritance is related to the typing concept. According to [14] a type is a predicate. An object is of a type if it satisfies the predicate. According to [23] every type generates an associated class, i.e. subtyping and subclassing go hand to hand. So, if there are two types *T1* and *T2* there must be the associated classes *C1* and *C2*.

We define a *superclass* to be the existing class and *subclass* to be the newly defined class 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.

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 of the redefined service conforms to the signature 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. According to [1] the redefinition of services must also obey two extra conditions:

1. The redefined service must return the same value as the original service in the superclass would 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.

Incremental inheritance provides the capability to allow objects to be specialized 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 do not have to guarantee the conformance of the signatures and the two above conditions imposed by behavioural inheritance.

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. Therefore, we believe that, in a specification, it is most useful if the distinction between behavioural and incremental inheritance hierarchies is dropped. Some object-oriented languages, such as Object Pascal, only allow the redefinition of a service, and therefore the use of the same service name, if the new service has exactly the same list of parameters and the same type of result (if the service is implemented as a function).

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 (associations). They are the most common type of relationships and are application dependent, 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", meaning that an "account has a card". These relationships are characterized 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. The lower bound specifies the minimum number of mappings between objects, and so it can define whether a relationship is optional or mandatory.

Some authors prefer to consider conceptual relationships to be defined implicitly in both directions, i.e. bidirectional [24], but others suggest that it is better to define each association in a single direction, i.e. unidirectional [15]. 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.

Binary Relationships

A binary relationship is defined between two objects, each one belonging to a different class. 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 zero or one elements of S_2 and the same condition holds for the inverse relation R^{-1}; it is called 1:N when each element of S_2 is related to zero or one elements of S_1 by the inverse relation and each element of S_1 can be related to zero or more elements of S_2; 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 zero or more elements of S_2, and in the R^{-1} relation, each element of S_2 can be related to zero or more 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.

Unary Relationships

Unary relationships involve two objects of the same class. 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 object models produced by [9, 15], for example, only define communication between objects of different classes.

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 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 [17, 18, 20].

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

- Composition, which combines objects to form larger objects;
- 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 most 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 Aggregation

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). An aggregate is, in any case, an object with identity.

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

In the first 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 this situation, the aggregate and each object components have their own identity.

In the second case, where the object components are not shared, the object components are *hidden* from the rest of the system and they will belong exclusively to the aggregate. Therefore, the interface with the rest of the system is made via the composite object and we may even decide to not give identity to each object component.

3.2.19 Subsystems

A subsystem is merely a grouping of objects and it should not be seen as an object since it has no identity.

Objects 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 OO 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.

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. Saving accounts give interest and cannot be accessed by the automatic tellers.

This problem has been analysed using OMT [24]. Part of the final object model is depicted in Figure 3.1.

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 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, non-deterministically, from one state to another by events defined in the body of the process definition which correspond to the services in the object model. Thus, the 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 2.2). The state information is

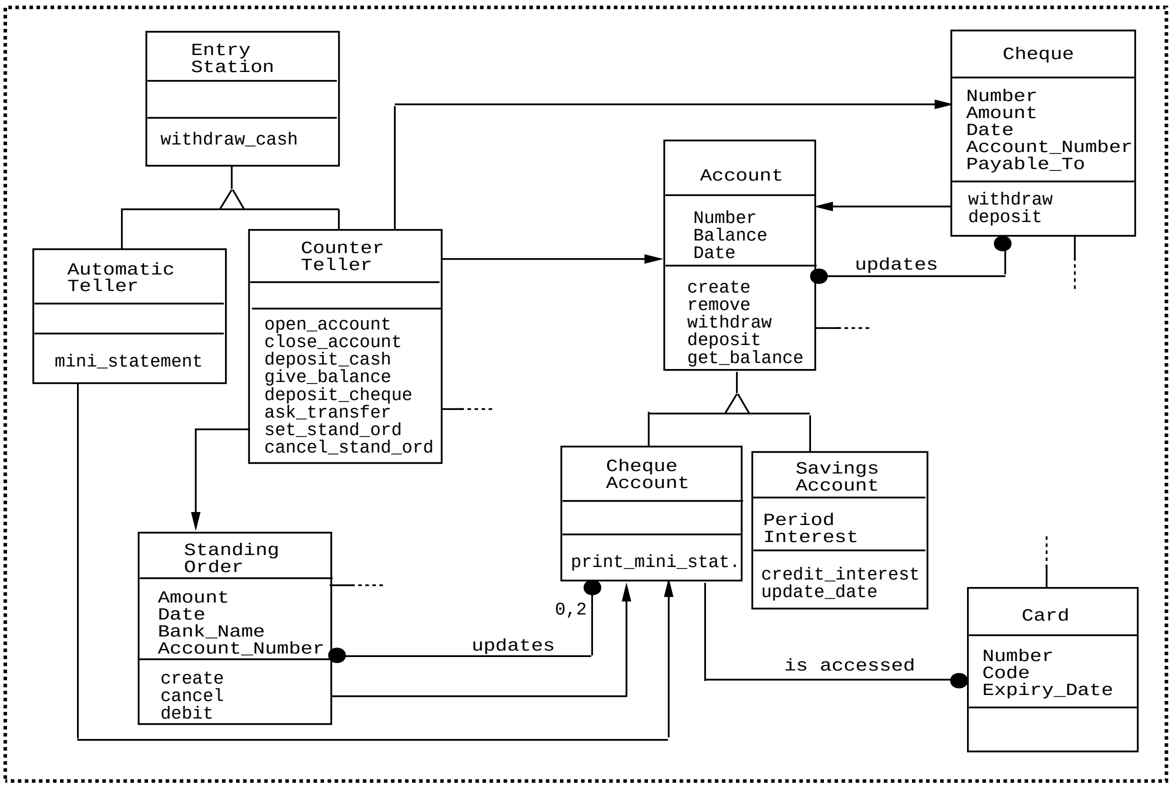


Figure 3.1: Part of the final object model

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.

The event

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

is a structured event where `g` is a gate, `selector 1` is the name of an offered service, and `Get Object 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.

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 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 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 initial state affected by the operation `F1`. The operation `F1` is defined in the ADT `State Template`.

Using the running example, let us consider the object model represented in Figure 3.1. 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 !withdraw_ok !Get_Account_Number(this_account) !true;
                   exit(Debit_Account(this_account, m))
    []
     [not (if_money)] -> g !withdraw_ok !Get_Account_Number(this_account) !false;
                         exit(this_account)
   )
  []
   ...

 ) >> accept update_account: State_Account in Account[g](update_account)
endproc
```

Notice that `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 functions `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 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 account (account has funds and account has no funds) without querying the account's balance and without doing any calculations.

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.

3.3.3 Symbolic ADTs

The arguments of a process are defined as ADTs. ADT definitions in LOTOS are usually long. Each operation is defined by one or more algebraic equations. Since we are in the analysis phase, it would be preferable to use ADTs where only a small number of simple equations are required. This can be achieved by using *symbolic* ADTs. A symbolic ADT contains only the necessary information to allow the specification 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. 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.

A symbolic ADT is built 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 we define 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 by using the template name followed by _Type, and let the sort name be the name of the template preceded by State_. The operations of the ADT can be named differently from the operations in the object model, or else with the same name followed by _ADT. The sort of auxiliary ADTs, such as Balance and Account_Number, can be named without the term State_. The constructor is always named with the template name preceded by Make_ and the name of the operation that gives the object identifier always start with Get_ and follows with the identifier name.

3.3.4 Services

The term service is used to denote an operation offered by both an object in an object model and a process. We reserve the term *operation* to denote the operations defined in 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 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. The operations in the ADT usually have a different name from the service name.

For example, consider the behaviour expression defined in template `Account`:

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

`withdraw` is the message name (it has the same name as the service in the object model) and `Debit_Account` is the name of the corresponding operation in the ADT `State_Account`.

The (first) action prefix `g !withdraw !Get_Account_Number(this_account) ?m: Money;` denotes the offered service `withdraw`. In order to perform an 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 choose between 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 very 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 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, but in order to accomplish them it may need to refer to other object's 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 operation defined in Account *)
```

3.3.5 Attributes

As noted earlier, the state information of the object is modelled in LOTOS as ADTs which appear as arguments in a template. It would be possible to model each attribute as a separate

ADT and then give all of them as parameters of the template. However, we prefer to put all the attributes together in a single ADT and then use it 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 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 which shows that an account has two attributes, `Account Number` and `Balance`. These two attributes are defined as two separate ADTs. A basic constructor will not always have all the attributes in its domain as some may be implicit.

3.3.6 Classes

A class is modelled in LOTOS as 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 processes `Object Generator` and `Template` are composed in parallel, but do not interact with one another.

The internal event `i` is used to control the recursive instantiation of `Object Generator` so that it is easier to follow the simulation using LITE (it is not necessary in the definition of the process).

In most real situations an object needs to be initialized when it is created and the initialization operation should only be offered once for each object. We could do that inside this template as follows [6]:

```
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 class. This is the way in which `create` should be regarded. Adopting this view, the definition of the class template is much simpler.

In the initialization 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 initialization 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 which is updated by a recursive call. 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_State(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.

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 has to offer an event such as

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

Synchronization takes place and `Template` is instantiated causing an object to be created with some state information. Note that while the values `val1, ...` are passed into `Object Generator`, the object identifier is created using value generation. There is also the situation where there is an upper limit to the number of objects. In this case, the object generator can hold the set of identifiers not yet allocated and the selection predicate guarantees that the new object identifier is one of these.

In the banking system, the 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 State(id1,...))
```

where `id1` is a specific identifier we give to the object.

3.3.7 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 simple object identifiers which can be used indistinguishably by any class of objects. This is achieved by defining a standard ADT and by adding it to the LOTOS library. Thus, we have defined the "standard" ADT `Id Type` with the following specification:

```
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
        First_Set, Second_Set, Third_Set, Fourth_Set : Id      -> Bool
        h : Id -> Nat
        _eq_, _ne_, _lt_ : Id, Id -> Bool
   eqns
     ...
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 3.3.11).

As noted earlier, the object generator in its extended form requires a set of identifiers. Therefore, `Set_Id_Type` has also been added to the library:

```
type Set_Id_Type is Set actualizedby Id_Type using
    sortnames Id   for Element
              Bool for FBool
endtype
```

The standard LOTOS library includes 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 predefined ADT for booleans.)

Whenever object identifiers are needed for a given class, `Set Id Type` is instantiated. For example, for the class of objects `Account`, we instantiate 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`, ..., `id20` 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 ODP standard model [14] 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 3.3.6, an account number would be generated when the object generator `Accounts` offers for synchronization:

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

and some other object, perhaps an interface object, offers:

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

When these two events synchronize, the same symbol (not a value) of the sort `Account Number` is ascribed to both `acc number` and `acc counter`. This symbol 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 available with the SMILE simulator [11]. The SMILE simulator is part of the Lite toolset produced as part of the Lotosphere ESPRIT Project. It allows us to simulate using a set of possible values rather than a particular value of the identifier.

3.3.8 Abstract Classes

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.

3.3.9 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 process 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 an unknown number of objects is required, the instantiation occurs indirectly by sending a create message to an object generator.
2. If the number of necessary objects is fixed, the template is instantiated directly.

In the banking example, as an unknown number of chequing accounts are needed, a new cheque account is created by means of the object generator `Cheque Accounts`. Therefore `Cheque_Accounts` offers:

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

and an instance of `Counter Teller` would offer:

```
g !create !cheque ?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 `Cheque Accounts` and each cheque account object would be created by calling the template `Cheque Account` directly, for example, `Cheque_Account[g](Make_Account(id1,0))`.

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 is best 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 sucessful termination of the process; and **stop**, representing the abnormal termination of the process. Therefore we use **stop**.

To delete an object, the operation `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. 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 events currently being offered by the object.

Behaviour (of an Object)

A *behaviour* describes the order in which the events of that object can occur and the changes in the object's state. Behaviours are required to define *sequencing rules*, the possible choice of events allowed at given time, and, for more complex behaviours, *concurrency rules* [14]. The state of an object can restrict the events that can occur within the object at a given time as events can have guards.

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 events which occur when the object synchronizes with other objects in the system. The internal, invisible behaviour is specified by events only defined internally to the object and so are invisible to the object's environment.

In summary, the behaviour of an object is given by a collection of events with a set of constraints on the order in which they may occur.

3.3.10 Message Connections

In LOTOS two or more processes communicate via event synchronization, 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 synchronize 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 a separate gate to communicate with each of 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 `Cheque Account` asking for an account balance:

```
g !get_balance !acc number ?balance:  Money;
```

and an instance of `Cheque 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" include 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 and a return event.

```
g !withdraw !acc_number !amount;
g !withdraw_ok !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 !withdraw_ok !Get_Account_Number(this_account) !true;
                exit(Debit_Account(this_account, m))
[]
  [not (if_money)] -> g !withdraw_ok !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.

However, it is possible that two objects belonging to the same class 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 defined as a process, which synchronizes with the process which defines the class template in question (see Figure 3.2).

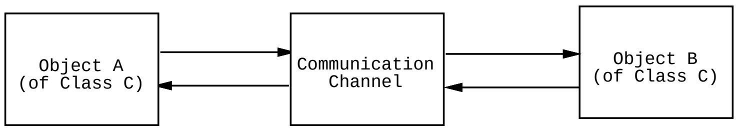


Figure 3.2: Two objects of the same class communicate via a channel

3.3.11 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. This document only deals with incremental inheritance, leaving for a future paper the discussion of both multiple inheritance and behavioural inheritance and on how the conditions given in Section 3.2 must be verified in a LOTOS specification. Pure extension, where there is no redefinition or deletion of services, does provide behavioural inheritance.

To be able to specify inheritance in LOTOS (extension and redefinition of operations and extension of attributes), the superclass has to be defined with **exit** functionality. 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
```

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) [22].

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 Inherited Class with the services we want to keep and use it as follows in the definition of the new class Redefined Class:

```
  process Inherited_Class[a](state: State_Sort) : exit(State_Sort) :=
    a !selector_1 !Get_Id(state) ... ;
    ...
    exit(state)
  []
    a !modifier_1 !Get_Id(state) ... ;
    ...
    exit(F1(state))
  endproc

  process Redefined_Class[a](state: State_Sort) : noexit :=
    ( ( Superclass[a](state)
      |[a]|
        Inherited_Class[a](state)
      )
        >> accept update_state: State_Sort in Redefined_Class[a](update_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` defined above, and supposing we want to define a subclass which extends the superclass state and redefines the service `modifier 2`, the process template `Inherited Class` would take the form:

```
  process Inherited_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
```

(Note that **any** is a LOTOS key word and can be used with any type, predefined or not).

And `Redefined Class` would be defined as follows:

```
  process Redefined_Class[a, b]
       (state: State_Sort, ext_state: Ext_State_sort) : noexit :=
    ( ( Superclass[a](state)
      |[a]|
        Inherited_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 `Redefined Class` is a pair of ADTs *(state, ext_state)*.

In the object model represented in Figure 3.1, `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 because it is a superclass its functionality has to be changed to **exit**:

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

The object generator of this template would be:

```
process Cheque_Accounts[g](accs: Account_Number_Set) : noexit :=
  g !create !cheque ?acc_counter: Account_Number
      [(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
```

As both kinds of account share the same `Account Number` sort, `!cheque` specifies 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`.

As we can see in the object model, the superclass does not have any specific instances, and so it does not need an object generator. If we however we change the requirements in order to allow the creation of objects of the superclass `Account`, a new process would have to be created:

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

3.3.12 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. If we are creating executable specifications, we have to say something about how 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 this association as a new object.

Any relationship involving a superclass will be inherited by the objects of its subclasses. Therefore such relationships can be modelled in the template that defines the superclass.

Binary Associations

One-To-One

A *one-to-one* association 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 this_account: State_Account in exit(this_account, cards)
  []
    g !print_mini_statement !Get_Account_Number(this_account) !this_account;
    exit(this_account, cards)
  []
    ...
  ) >> accept this_account: State_Account, cards: Card_Number_Set
    in Cheque_Account[g](this_account, cards)
endproc
```

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

```
process Cheque_Accounts[g](accs: Account_Number_Set): noexit :=
  g !create !cheque ?acc_counter: Account_Number
     [(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 initializing 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](accs: Account_Number_Set) : noexit :=
  hide cd in
  ( g  !create !cheque ?acc_counter: Account_Number
       [(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. 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 3.1, 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 1:1 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. However, for simplicity, at this stage, we model it as an attribute that is a set of object identifiers in each object.

In the example, `Cheque Account` has a many-to-many relationship with `Standing Order`. We must add to `Cheque Account` a 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, g](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, g](Make_SO(so_counter, n1, n2, bk, m))
  |||
    Standing_Orders[a, g](Insert(so_counter, sos))
  )
endproc
```

Unary Associations

Unary associations 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) associations. 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/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 following:

```
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.

3.3.13 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.

3.3.14 Aggregation

In cases where the object components are shared with other objects in the system, aggregation is treated in the same way as conceptual relationships.

The case where the object components are not shared needs further development.

3.3.15 Subsystems

A subsystem is a collection of objects 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 while a subsystem is not and therefore it has no identifier.

We can use the LOTOS interleaving and parallel operators to form a subsystem.

In our banking system, we create the subsystem `Complex Operations`, built from the classes `Cheque` and `Standing Order`:

```
process Complex_Operations[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.

Chapter 4

The Rigorous Object-Oriented Analysis Method

4.1 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. Formalizes the object model describing each class template and relationships in a mathematical manner.
2. Adds state information and behaviour to each object 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 formalization of the object model can be done semi-automatically, i.e. it requires some decisions, but most of the work is fairly straightforward and 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. Standard OOA methods propose two extra models (see Figure 4.1) 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.

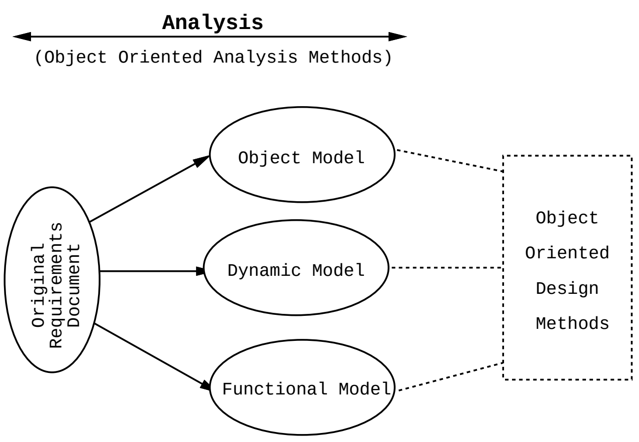

Analysis
(Object Oriented Analysis Methods)
Original Requirements Document
Object Model
Dynamic Model
Functional Model
Object Oriented Design Methods

Figure 4.1: The models built by many object-oriented analysis methods

4.2 The ROOA Method

ROOA gives us an integrated view of the system, showing at the same time its static and dynamic properties, and the information exchanged during communication. 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:

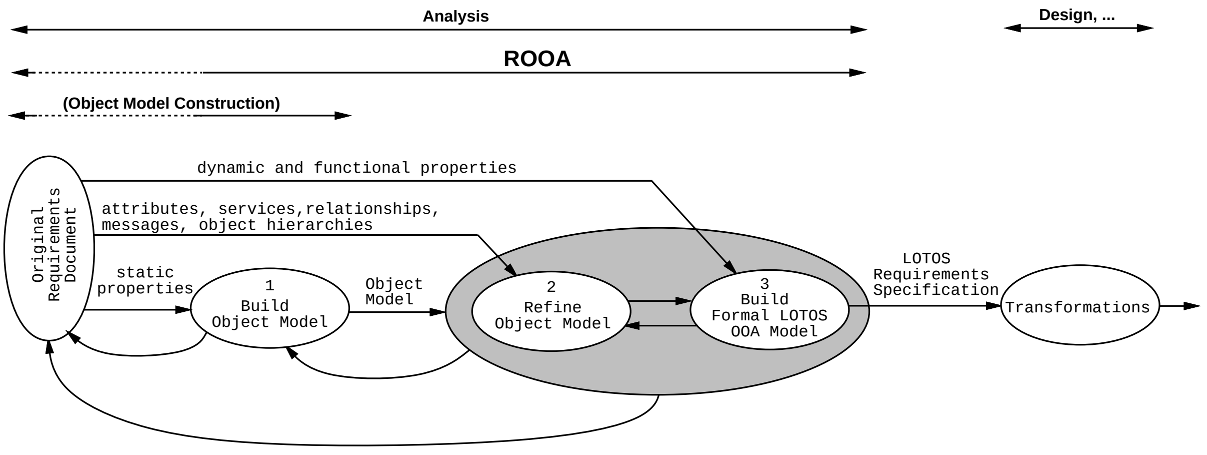
1. Build an object model.
   The construction of the object model is performed by applying any of the existing object-oriented analysis methods, such as [9, 15, 24].

2. Refine the object model.
   We refine the object model by guaranteeing that it includes interface objects, attributes, services, static relationships and message connections, and 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 4.2 illustrates ROOA in the context of the software development life cycle and shows how the various tasks are connected.

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

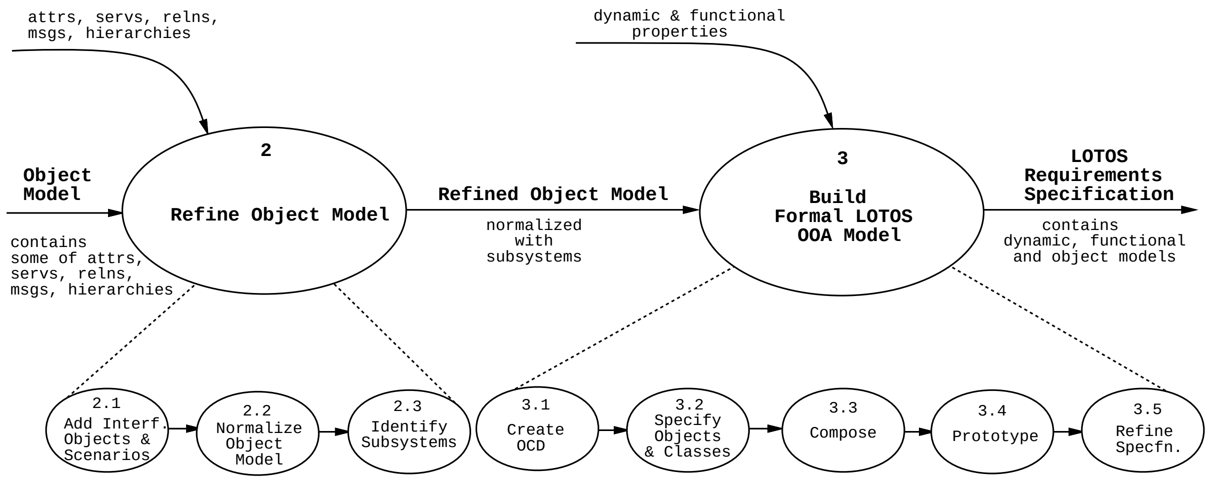


Analysis
Design, ...
ROOA
(Object Model Construction)
Original Requirements Document
dynamic and functional properties
attributes, services,relationships, messages, object hierarchies
static properties
1 Build Object Model
Object Model
2 Refine Object Model
3 Build Formal LOTOS OOA Model
LOTOS Requirements Specification
Transformations

Figure 4.2: Context of ROOA in the development life cycle

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. We can also adopt a trajectory where the resulting requirements specification is transformed into a design specification with prototyping being used to ensure that the two specifications conform to one another.

Figure 4.3 gives a view of the two main tasks of ROOA, *Refine Object Model* and *Build Formal LOTOS OOA Model*, showing their composition.



attrs, servs, relns, msgs, hierarchies
dynamic & functional properties
Object Model
contains some of attrs, servs, relns, msgs, hierarchies
2 Refine Object Model
Refined Object Model
normalized with subsystems
3 Build Formal LOTOS OOA Model
LOTOS Requirements Specification
contains dynamic, functional and object models
2.1 Add Interf. Objects & Scenarios
2.2 Normalize Object Model
2.3 Identify Subsystems
3.1 Create OCD
3.2 Specify Objects & Classes
3.3 Compose
3.4 Prototype
3.5 Refine Specfn.

Figure 4.3: Core of ROOA

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 formalize it. It then has to identify the dynamic properties of each object 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.

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 what 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 more than 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 [11] for their work, since it gives value generation and has a lot of other very useful characteristics. However, this tool is perhaps too complicated to be used to show the system to the client. The simulator SEDOS [26] does not offer so 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.

4.3 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 as well as the static relationships and typically involves many instances of each class template. So that we do not become 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 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.

An example of an object model for the problem described in Section 3.3 and which was built by using OMT [24] is depicted in Figure 4.4.

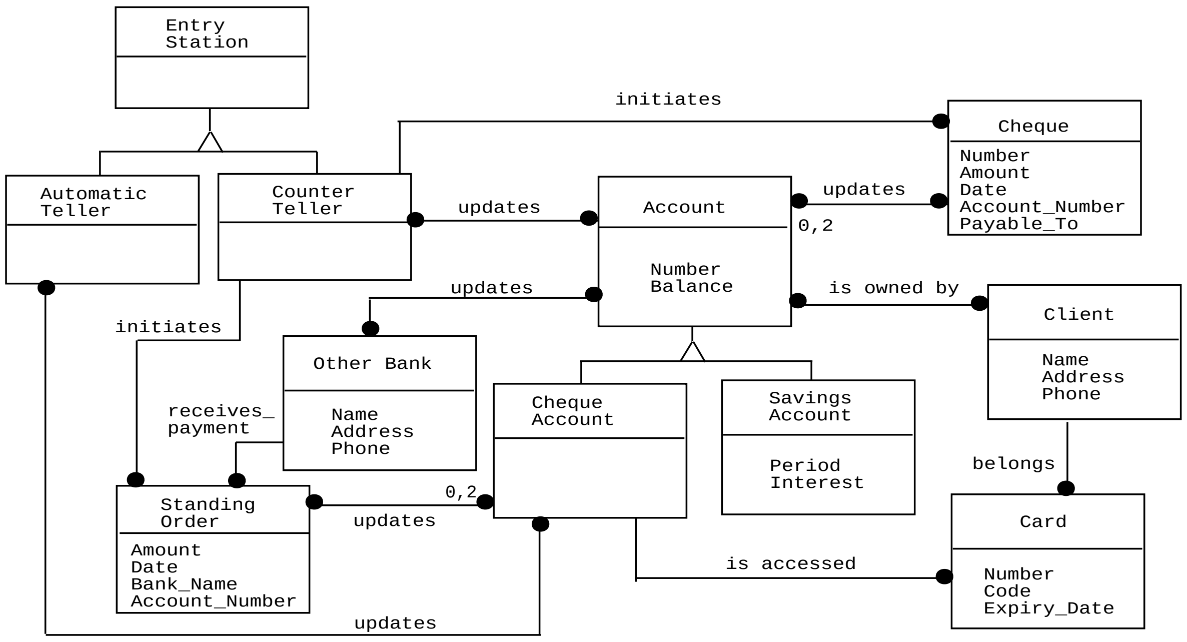


Figure 4.4: Object model produced by the OMT method

Task 2: Refine the Object Model

The object model gives the static view of a problem. The kind of object which is described and the information it includes depend upon the method used.

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 [15], there are however other kinds of objects: *interface objects* and *control objects*. Interface objects are the objects that the *actors* use to communicate with, ask for services and receive answers from, the system. (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. (The control objects may be dealt with in a later stage.)

In object models created using some OOA methods, the objects are defined only by name and a list of attributes, in others we also have services. We propose that an object can only be fully understood if it is defined by a name, a list of attributes and a list of services offered by that object. Moreover, in our object model the objects are related by static relationships and message connections. This is achieved by normalizing the object model.

During this task, we also start structuring the system. There are OOA methods that propose subjects [9], subsystems [15], or modules [24] to group the objects in the object model. 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.

Task 2.1: Add Interface Objects and Define Interface Scenarios

An interface object models 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`.

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.

Notice that interface objects belong to the system, but interface scenarios do not.

Task: Add interface objects to the object model and define interface scenarios.

Task 2.2: Normalize the Object Model

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 *normalizing* the object model, i.e. by guaranteeing that it incorporates the following set of properties:

1. Static relationships between objects.
2. Attributes and services in each object.
3. Message connections between objects.

Task 2.2.1: 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. Aggregates.
2. Inheritance.
3. Conceptual relationships.

Relationships are present in most object models given by OOA methods. However, the EVB method [2] 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 2.2.2: 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 (the environment of the object) 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 [15, 24]. 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.

During this task, 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 are only building the first two columns. In the first column we list the objects that form the object model and in the second column we list the services offered by each object. If the services are given in the object model, column one and column two can be filled in directly (see Table 4.1).

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.

Methods such as [9] define objects with attributes and services.

Task: Add attributes or services, as appropriate, to the objects in the object model. Fill columns one and two of the OCT.

Task 2.2.3: Message Connections

A message connection shows a processing (dynamic) dependency between a client object and its server. Message connections are defined as single arrows, not double arrows. Therefore, if object *Obj_A* needs object *Obj B* and object *Obj B* needs object *Obj A*, we draw two arrows, as shown in Figure 4.5.

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 an object in column one, the services that that object requires from other objects to accomplish that particular service. The notation <*object.service*> is used to refer to the required *service* defined in *object*. In the fourth column (*Clients*) we list,

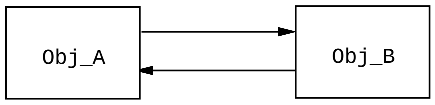


Figure 4.5: Message connections between objects A and B

for each service offered in column one, the objects (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 interface scenarios. 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 the objects 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 object 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.

For the object model in Figure 4.4, and considering each object with its services, the first four columns of the OCT would be filled as shown in Table 4.1 and the information added to the object model, as shown in Figure 4.6.

While 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 objects, 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 object `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. This information will be given later on by LOTOS, but we could use message sequence charts to describe it and then use them as a guideline to build the LOTOS specification.

The message connections are drawn in the object model according to the OCT. If one object requires services defined in more than one object (server), then there is an arrow starting from the first object reaching each of the servers. Only one message connection is drawn from one particular object to another object, independently of the number of services the first object 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: Identify Groupings of Objects

Grouping objects into subsystems or into composite objects 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 objects 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 object increases.

Objects	**Offered Services** (by the server)	**Required Services** (from another object)	**Clients** (of each offered service)
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	Interface Scenario
		Account.deposit	
		OB.send_transfer	
	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	internal
		Account.deposit	
		OB.send_transfer	
Cheque	withdraw	Account.withdraw	CT
	deposit	Account.withdraw	CT
		Account.deposit	
		OB.cheque_withdraw	
Account (A)	create		CT
	remove		CT
	deposit		CT, Cheque, OB, SO
	withdraw		CT, ES, Cheque, OB, SO
	get_balance		CT
Cheque_Account (CA)	print mini stat.		AT
Savings_Account (SA)	credit interest		internal
	update_date		internal

Table 4.1: OCT with objects, services offered, services required and clients

During the first iteration, only *obvious groupings* are identified. Suitable candidates are:

1. Objects that participate in an aggregate relationship. They form a composite object.
2. Objects that participate in an inheritance relationship. They form a subsystem.

The objects that compose a subsystem or a composite object are known as object components (or class components). Grouping other objects should be left until later iterations (rules are given in Task 3.5.3).

We show groupings in the object model by surrounding the objects by a rectangle with dotted lines. The OCT should also be changed to encode this information. The changes include using the name of the subsystem or composite object in the first column of the table, instead of the object component's name, and using the name of the subsystem or composite object followed by the name of the object component between brackets in any other place where that object 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 4.2.

Figure 4.6 represents the refined object model first shown in Figure 4.4.

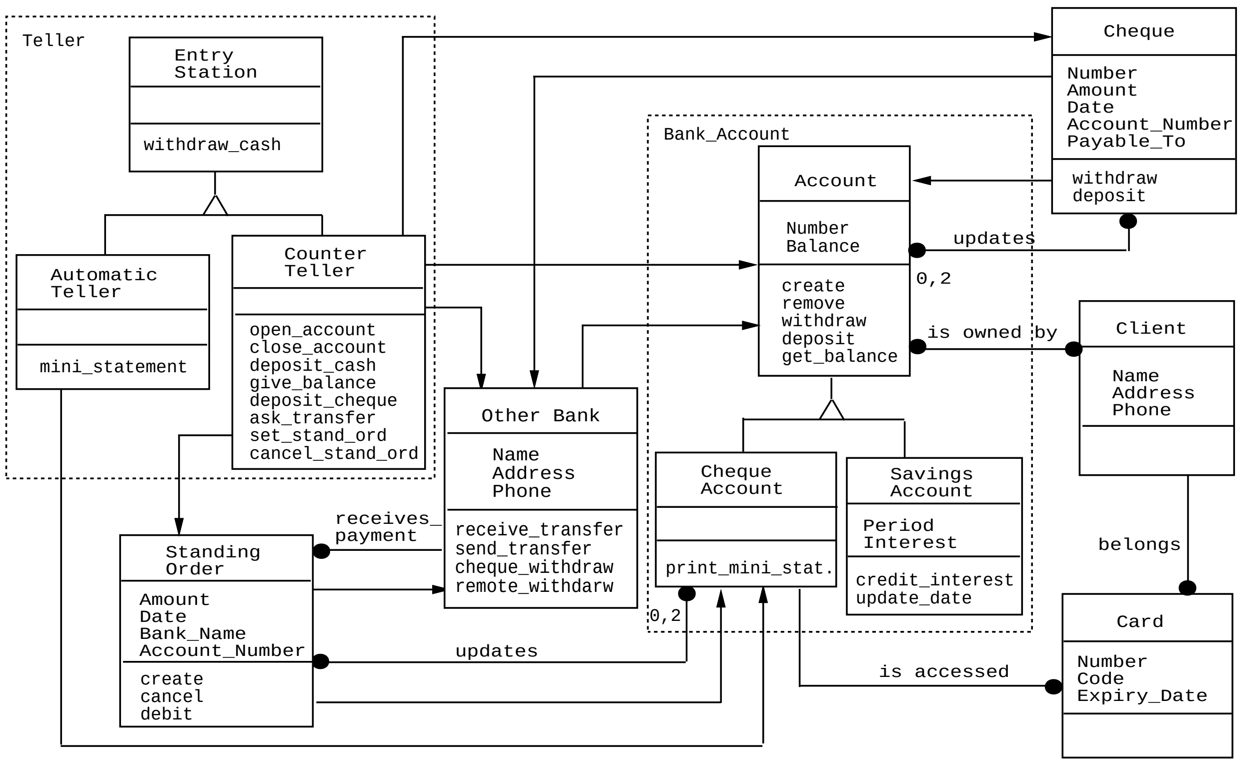


Figure 4.6: Refined object model

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.

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 OCD is an intermediary model between the object model and the LOTOS specification and it reflects the exact structure of the LOTOS formal 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 objects 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.
     Table 4.2 shows the gate names for the banking example.

   (b) Give gate names to the arcs in the OCD as follows:
   For each object in column one of the OCT, identify the arc in the OCD that connects this 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.

The initial OCD built by following the above rules is depicted in Figure 4.7.

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 [9].)

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.

Objects	**Offered Services** (by the server)	**Required Services** (from another object)	**Clients** (of each offered service)	**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	Interface Scenario	t
		BA.deposit		
		OB.send_transfer		
	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	internal	
		BA.deposit		
		OB.send_transfer		
Cheque	withdraw	BA.withdraw	Teller(CT)	c
	deposit	BA.withdraw	Teller(CT)	c
		BA.deposit		
		OB.cheque_withdraw		
Bank_Account (BA)	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
	credit_interest(SA)		internal	
	update_date(SA)		internal	

Table 4.2: OCT with gates

In the later iterations object generators and new groupings are added to the model. This requires the following changes in the diagram:

6. Introduce object generators.
   In later iterations, object generators are introduced. After this, 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.

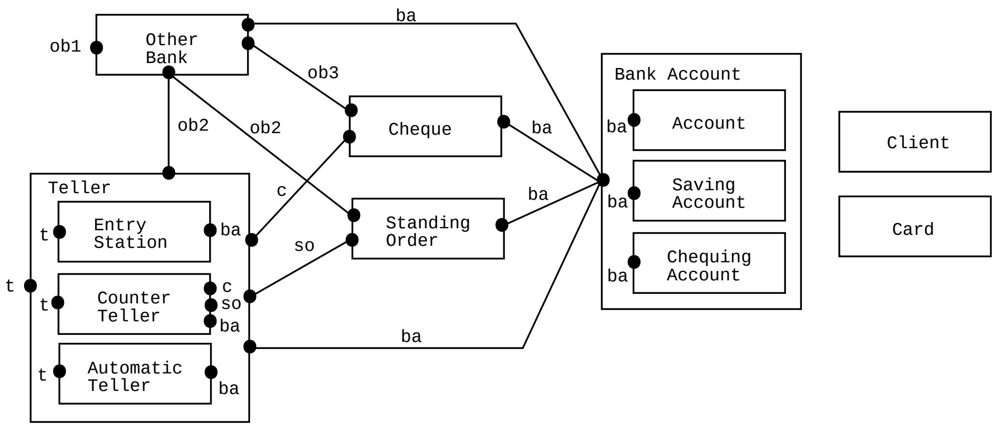


Figure 4.7: Initial object communication diagram

7. Introduce new groupings of objects.

   New grouping 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 object component between brackets after the name of the service in column two (see Tables 4.2 and 4.3.) 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 object 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 an object component with the name of its subsystem or composite object. 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.

In Table 4.3 we show the final OCT dealing with `Complex Service` which was built from `Cheque` and `Standing Order`.

The final OCD, which is based on Table 4.3 and corresponds to the refined object model presented in Figure 4.6, is given in Figure 4.8.

Task 3.2: Specify Individual Objects and Classes

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.

Objects	Offered Service (by the server)	Required Service (from another object)	Clients (of each offered service)	Gates
Teller	open_account(CT)	BA.create	Interface Scenario	t
	...			
Other_Bank (OB)	receive_transfer	BA.deposit	Interface Scenario	ob1
	send_transfer		Teller(CT), CO(SO)	ob2
	cheque_withdraw		CO(Cheque)	ob2
	remote_withdraw	BA.withdraw	Interface Scenario	ob1
Complex_Operation	create(SO)		Teller(CT)	cs
(CO)	cancel(SO)		Teller(CT)	cs
[SO + Cheque]	debit(SO)	BA.withdraw	internal	
		BA.deposit		
		OB.send_transfer		
	withdraw(Cheque)	BA.withdraw	Teller(CT)	cs
	deposit(Cheque)	BA.withdraw	Teller(CT)	cs
		BA.deposit		
		OB.cheque_withdraw		
Bank_Account (BA)	create		Teller(CT)	ba
	remove		Teller(CT)	ba
	deposit		Teller(AT,CT), OB, CO(Cheque, SO)	ba
	withdraw		Teller(ES,CT), OB, CO(Cheque, SO)	ba
	get_balance		Teller(CT)	ba
	...			

Table 4.3: Final OCT

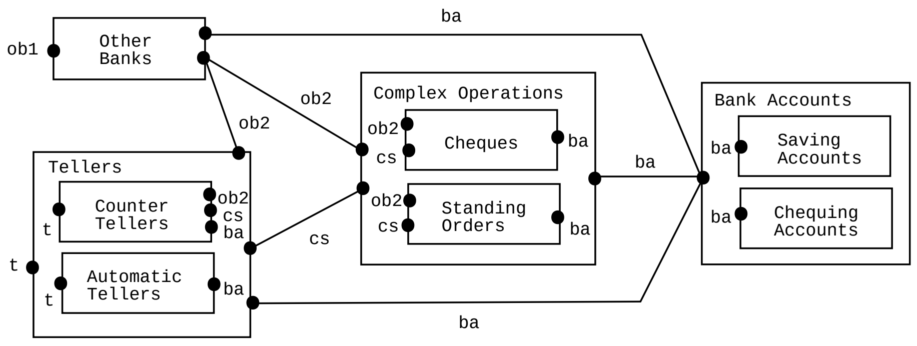


Figure 4.8: Final object communication diagram

In Chapter 3 we discussed how to model objects, class templates and classes in LOTOS. 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 symbolic 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 symbolic ADTs using ACT ONE.

A process definition with its ADTs fully specifies an object, i.e. defines the class template. It defines the static and dynamic aspects of an object. The object model defines each object by its name, a list of services and a list of attributes. An ADT alone could define and formalize the information. The behaviour of each object is not, however, given by the object model. The OOA methods give it in a separate model, the dynamic model. A process, 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 each static relationship in the object model. Finally, we compose the processes into behaviour expressions, defining in this way the behaviour of the whole system.

In addition to the advantages enumerated above, we end up with an executable specification where we can use rapid prototyping.

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 an object we take as a starting point columns two and three of the OCT. Column two gives the services offered by the object and column three gives the services required by that object, from another object. Then, we have to find out the order in which the calls to the required services occur. For that, we should imagine ourselves inside the object 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 the offered services 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.

The events are modelled in LOTOS as structured events, as discussed in Section 3.3.10.

Specifying an object as a process, we show the events 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 object in terms of its attributes. Comparing the object defined with a process and ADTs with the object 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 and dynamic properties, and the information passed in a formal manner.

In order to specify a process incrementally, only a subset of events may be dealt with in the first iteration.

Task 3.2.2: Specify 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 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.

The order in which we specify processes and ADTs (Tasks 3.2.1 and 3.2.2) is not fixed. 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.

Notice that 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 normalized 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 in [7] 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 4.7 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]
```

In later iterations, when object generators are introduced to deal with multiple instances, the composed behaviour expression is refined and built 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 and message connections. 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).

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 objects. 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 object component, or else define the object generator for a composite object. Which is to be preferred depends on each particular situation.

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 objects (subsystems or composite objects) 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 `Complex_Operations`. These changes can be seen in the OCT represented in Table 4.3 and in the OCD in Figure 4.8. The composition of the objects in the OCT would now take the form:

```
( 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]
```

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 2.3 and 3.

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.

4.4 The ROOA Documents

The most useful form of describing a process is in terms of work products [21]. ROOA is not only a process of developing software. It also produces documentation as the process is applied. In Figure 4.9 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

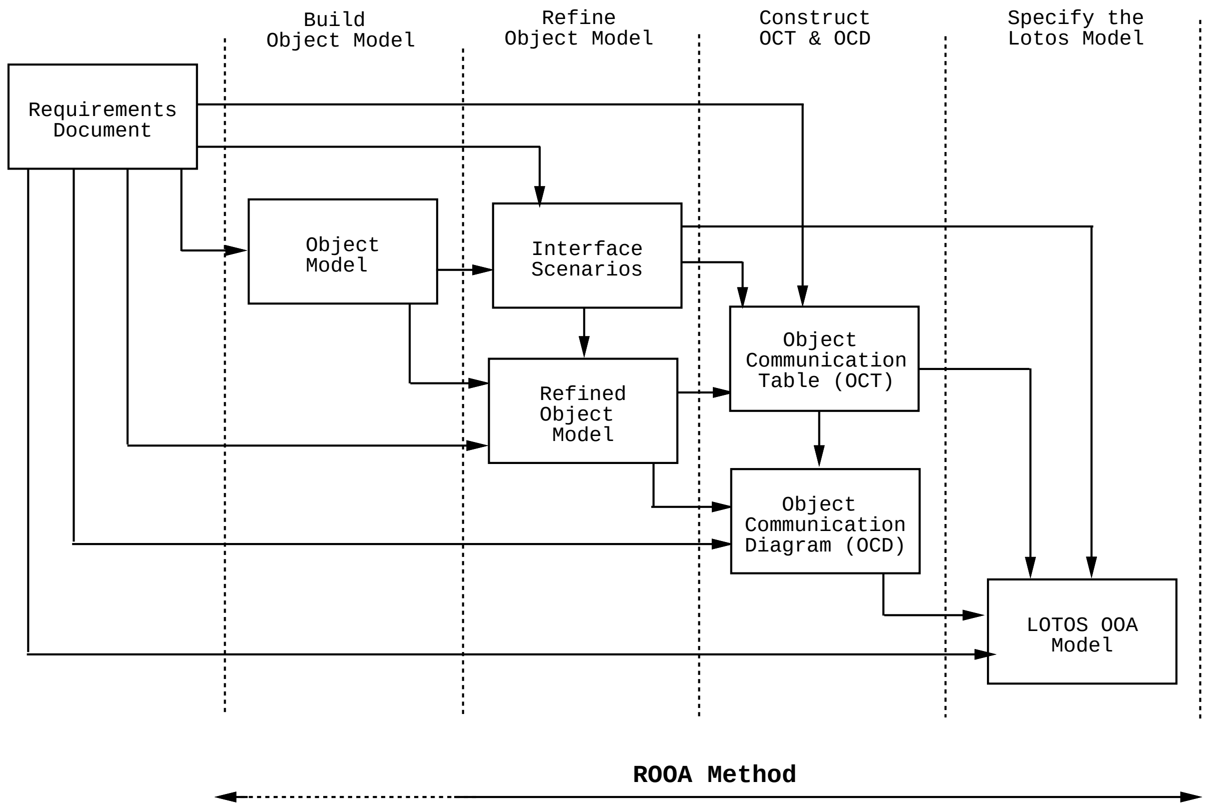


Figure 4.9: Documents produced during the application of ROOA

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.

Chapter 5

Conclusions and Prospects

5.1 Conclusions

We have presented the ROOA (Rigorous Object-Oriented Analysis) method. It enables a formal object-oriented analysis model to be devised from informal requirements, and results in a formal requirements specification expressed in the formal description technique LOTOS. ROOA 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 clients.

We have described the representation of standard object-oriented analysis concepts in LOTOS. This includes the representation of objects as LOTOS processes with symbolic ADTs. The LOTOS process specifies the dynamic behaviour of the object and the ADTs, 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 ADT. A symbolic ADT 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. A class template is represented as a LOTOS 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. An object identifier is defined by a LOTOS ADT, 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 synchronization*, in which two processes synchronize on a *gate*, and may exchange data, represented by values. 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 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. We have mentioned two techniques for grouping objects: composition and decomposition. We have shown how composition may be achieved using the LOTOS parallel composition operators.

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 in that part 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 standard object-oriented analysis methods. The object model is refined in the second task by passing through three subtasks: adding the interface objects and defining interface scenarios, normalizing the object model and identifying object groupings. The formal LOTOS OOA model is built by integrating the dynamic and functional properties of the system into the refined object model, and consists of five subtasks: creating the object communication diagram (OCD); specifying the objects and classes 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.

We have illustrated this method on a small running example, representing a simple automated banking system.

5.2 Future Work

The development of ROOA is not complete. In particular, we need to devise rules for grouping objects to form subsystems and develop the concept of communication between objects of the same class. More work is also required on aggregates and on multiple and behavioural inheritance.

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 of no use 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.

Finally, we have not yet investigated the optimization of the various techniques within ROOA. For example, we do not yet have an algorithm for minimizing the number of gates in the object communication diagram. These issues are subjects for further research work.

5.3 Acknowledgments

We would like to thank Charles Rattray and Peter Ladkin for their helpful comments on an earlier draft of this report.

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

Bibliography

[1] T. Bar-David. Practical Consequences of Formal Definitions of Inheritance. *Journal of Object-Oriented Programming*, 5(4):43–49, July/August 1992.

[2] E. Berard. Object Oriented Requirements Analysis. EVB Software Engineering Inc., 1989.

[3] S. Black. Objects and LOTOS. Technical report, Hewlett-Packard Laboratories, Stoke Gifford, Bristol, 1989.

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

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

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

[14] ISO. Basic Reference Model of Open Distributed Processing. ISO/IEC JTC1/SC21, American National Standards Institute, 1994. Draft standard.

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

[16] T. Mayr. Specification of Object-Oriented Systems in LOTOS. In K.J. Turner, editor, *Formal Description Techniques*, pages 107–119. North-Holland, 1989.

[17] 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.

[18] G.A. Miller. The Magical Number Seven After Fifteen Years. *Studies in Long Term Memory*, 1975.

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

[20] D.L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 5(12):1053–1058, December 1972.

[21] 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.

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

[23] S. Rudkin. Templates, Types and Classes in Open Distributed Processing. *BT Technology Journal*, 11(3):32–40, July 1993.

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

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

[26] 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.