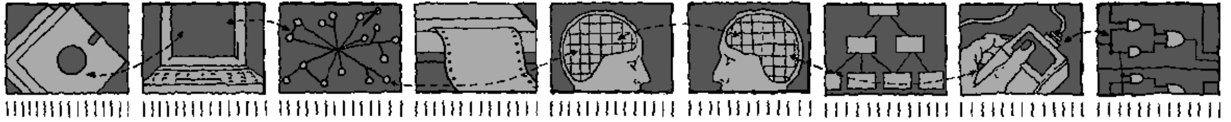
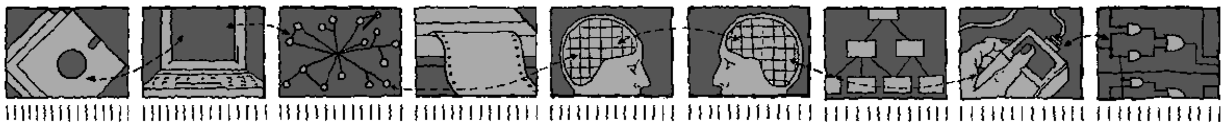


*Department of Computing Science and Mathematics*
*University of Stirling*



# Specification Case Studies in ROOA

**Ana M. D. Moreira and Robert G. Clark**

*Technical Report CSM-129*

October 1994

*Department of Computing Science and Mathematics*
*University of Stirling*

# Specification Case Studies in ROOA

**Ana M. D. Moreira and Robert G. Clark**

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

Telephone +44-786-467421, Facsimile +44-786-464551
Email amm@fct.unl.pt | rgc@compsci.stirling.ac.uk

*Technical Report CSM-129*

October 1994

# Contents

**Abstract** **v**

**1 Introduction** **1**

**2 Automated banking system** **3**
2.1 Introduction . . . . . 3
2.2 The solution with OMT . . . . . 3
2.2.1 The object model . . . . . 4
2.2.2 The dynamic model . . . . . 9
2.2.3 The functional model . . . . . 14
2.3 Applying ROOA . . . . . 19
Task 1: Build an object model . . . . . 19
Task 2: Refine the object model . . . . . 19
Task 3: Build the LOTOS formal model . . . . . 20
Task 3.1: Create an Object Communication Diagram (OCD) . . . . . 21
Task 3.2: Specify class templates . . . . . 21
Task 3.3: Compose the objects into a behaviour expression . . . . . 25
Task 3.4: Prototype the specification . . . . . 26
Task 3.5: Refine the specification . . . . . 27
Task 3.5.1: Model static relationships . . . . . 27
Task 3.5.2: Introduce object generators . . . . . 27
Task 3.5.3: Identify new higher level objects . . . . . 29
Task 3.5.4 and 3.5.5: Demote and promote an object to be specified as ADTs or as processes . . . . . 30
Task 3.5.6: Refine processes and ADTs . . . . . 30
2.4 Conclusions . . . . . 31

**3 Warehouse management system** **33**
3.1 Introduction . . . . . 33
3.2 Applying ROOA . . . . . 33
Task 1: Build an object model . . . . . 33
Task 2: Refine the object model . . . . . 34
Task 3: Build the LOTOS formal model . . . . . 37
Task 3.1: Create an Object Communication Diagram (OCD) . . . . . 37
Task 3.2: Specify class templates . . . . . 38
Task 3.3: Compose the objects into a behaviour expression . . . . . 43
Task 3.4: Prototype the specification . . . . . 43
Task 3.5: Refine the specification . . . . . 44
3.3 Conclusions . . . . . 45

**4 A car rental system** **47**
4.1 Introduction . . . 47
4.2 The car rental system requirements . . . 47
4.3 Applying ROOA . . . 48
Task 1: Build an object model . . . 48
Task 2: Refine the object model . . . 48
Task 2.1: Complete the o bject model . . . 49
Task 2.2: Initial identification of dynamic behaviour . . . 49
Task 2.3: Structure the object model . . . 50
Task 3: Build the LOTOS formal model . . . 51
Task 3.1: Create an Object Communication Diagram (OCD) . . . 51
Task 3.2: Specify class templates . . . 52
Task 3.3: Compose the objects into a behaviour expression . . . 53
Task 3.4: Prototype the specification . . . 53
Task 3.5: Refine the specification . . . 54
4.4 Conclusions . . . 54

**5 Conclusions** **55**

**Bibliography** **57**

**A LOTOS specification for the banking system** **59**

**B LOTOS specification for Warehouse management system** **77**

**C LOTOS specification for the car rental system** **89**

# List of Figures

2.1 Initial approach to an object model diagram . . . . . 7
2.2 Object model with attributes . . . . . 8
2.3 Object model with inheritance . . . . . 9
2.4 Event trace diagram for automatic teller . . . . . 11
2.5 Event trace diagram for deposit cheque initiated by CT . . . . . 11
2.6 Event trace diagram for transfer initiated by CT . . . . . 11
2.7 Event flow diagram for automatic teller . . . . . 12
2.8 State diagram for automatic teller . . . . . 12
2.9 State diagram for counter teller . . . . . 13
2.10 State diagram for cheque . . . . . 13
2.11 State diagram for `cheque has funds` . . . . . 14
2.12 Input and output values for the automated banking application . . . . . 15
2.13 Data flow diagram for automated banking application . . . . . 15
2.14 Data flow diagram for `accept transaction` . . . . . 16
2.15 Data flow diagram for `perform transaction` . . . . . 16
2.16 Data flow diagram for `deposit cheque` . . . . . 17
2.17 Final object model . . . . . 21
2.18 Initial OCD . . . . . 22
2.19 Final OCD . . . . . 29

3.1 Simplified object analysis diagram produced by Jacobson . . . . . 34
3.2 Initiate enter new item . . . . . 35
3.3 Execute enter new item . . . . . 35
3.4 Initiate remove item . . . . . 36
3.5 Remove item . . . . . 36
3.6 Initiate redistribution . . . . . 36
3.7 Execute redistribution . . . . . 37
3.8 Warehouse object model . . . . . 40
3.9 Initial OCD for the warehouse system . . . . . 40
3.10 Final OCD . . . . . 44

4.1 Initial object model . . . . . 48
4.2 Event trace diagram for reservation transaction . . . . . 49
4.3 Event trace diagram for rental transaction . . . . . 50
4.4 Event trace diagram for return transaction . . . . . 50
4.5 Final object model . . . . . 52
4.6 An OCD for the car rental system . . . . . 52

# List of Tables


2.1 Candidates for objects . . . . . 4
2.2 Final list of objects . . . . . 5
2.3 Candidates for Associations . . . . . 6
2.4 Final list of associations . . . . . 7
2.5 Automatic teller machine scenario . . . . . 10
2.6 Deposit cheque scenario . . . . . 10
2.7 Transfer service scenario . . . . . 10
2.8 Function description for `withdraw (cash or cheque)` . . . . . 17
2.9 OCT with class templates, services offered, services required and clients . . . . . 20
2.10 OCT with gates . . . . . 22
2.11 Refined OCT . . . . . 29

3.1 The OCT for the warehouse system . . . . . 38
3.2 The OCT for the warehouse system (continued) . . . . . 39

4.1 Interface scenario for Reservation . . . . . 49
4.2 An OCT for the car rental system . . . . . 51

# Abstract

The Rigorous Object-Oriented Analysis (ROOA) method [11, 12, 13, 14, 16] starts from a set of informal requirements and produces a formal object-oriented analysis model that acts as a requirements specification. This specification is expressed in the standard formal description language LOTOS [1, 2].

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

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

ROOA has been applied to several problems. This document shows how this can be done, by presenting three case studies: an automated banking system, a warehouse management system and a car rental system.

# Chapter 1

# Introduction

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

We have developed the Rigorous Object-Oriented Analysis (ROOA) method [11, 12, 13, 14, 16] which specifies the required behaviour of a system by constructing a model using the formal description technique LOTOS (Language Of Temporal Ordering Specification) [1, 8]. As LOTOS has a formal semantics, the model has a precise meaning and can be used as a formal requirements specification of the system's intended behaviour.

The ROOA method shows how LOTOS can be integrated with object-oriented analysis methods. ROOA complements existing object-oriented analysis methods (such as those by Rumbaugh *et al.*, Coad and Yourdon, Jacobson and Shlaer-Mellor), enabling precision and formality in development where required, for example in safety-sensitive systems [5, 9, 17, 18].

The model produced by ROOA integrates the static, dynamic and functional models, unlike informal object-oriented analysis methods, such as the one by Rumbaugh *et al.*, which create three separate models [17]. It is primarily a dynamic model, but it keeps the structure of the static object model. The object communication table and the object communication diagram are intermediate structures which help us in the difficult transformation from the static to the dynamic model and in tracing the requirements through to the final formal model.

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

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.

In this document we show how ROOA can be applied to create a formal requirements specification. We analyse three problems: an automated banking system, a warehouse management system and a car rental system. The automated banking system is discussed in more detail than the other two examples. For this example, we use the OMT method [17] to build an object model, a dynamic model and a functional model. ROOA only needs an object model, but for this exercise we decided to show how ROOA would behave if the three models already exist. For the second example, we start by using the object model developed by Jacobson [9]. After some attempts to understand this model, we ended making some modifications before applying ROOA. Finally, the third example describes a simple car rental system.

While in the first example inheritance plays an important role, in the second example aggrega-

tion is the interesting object-oriented concept. The specification of dynamic behaviour also played a much more important role than it did in the banking example. The associated LOTOS specifications are given in Appendices A and B. The LOTOS specification for the car rental system is given in Appendix C, but only the more interesting functionality of the system is explored.

The purpose of this report is to show the application of the ROOA method. A full description of the method is given in [11].

# Chapter 2

# Automated banking system

## 2.1 Introduction

In this chapter we first apply the OMT method to create an object model, a dynamic model and part of a functional model. This shows how we can apply ROOA in situations where an OOA method has already been applied.

In the following chapters we only apply OOA methods to create an object model, and from there start applying the ROOA tasks.

The system we are specifying is to implement an automated banking application by managing the transactions the clients do with their own accounts either by interacting with automatic teller machines, by writing cheques or by dealing directly with the counter tellers. The requirements of the problem are as follows:

> The clients can take money from their own accounts, can deposit money, and can ask for their current balance. All these operations can be accomplished either directly in the bank or by using automatic teller machines. Withdrawing money from an account can be done by cheque, by transfer to other banks, or by card, using the teller machine. It is also possible to withdraw money from one account if the client authorizes a standing order. Depositing money in an account can be done by cheque from another account, by transfer from other banks or by giving actual cash to the counter tellers. In order to access an automatic teller machine, the client has a key card with a special code. The teller machine allows the client both to withdraw and to get the balance of an account. There are two kinds of accounts: savings accounts and cheque accounts. Savings accounts give a periodical interest and cannot be accessed by the automatic tellers. Cheque accounts can either be updated by the automatic teller machines or counter tellers. This implies that key cards are only associated with cheque accounts.

## 2.2 The solution with OMT

The Object-Oriented Modelling Technique (OMT) method proposed by Rumbaugh *et al.* [17] incorporates three models: the object model, the dynamic model and the functional model. The object model shows the objects and their static relationships; the dynamic model shows the order in which the operations are performed; the functional model shows the data transformations in the system. The relative importance of each model depends upon the kind of problem being analysed.

ROOA formalises the object model produced by OMT and then integrates the result with a LOTOS behaviour model. Nevertheless, we decided to present here a complete solution using Rumbaugh's method, with the Rumbaugh object, dynamic and functional models, in order to show the advantages obtained when LOTOS is used with another OOA method.

We start by constructing the object model, then the dynamic model and finally the functional model.

### 2.2.1 The object model

The object model gives the static structure of the system. This model describes the objects in the system and their static relationships which include the generalization relationship (inheritance). OMT proposes an enhanced entity relationship diagram to represent the object model. The enhancements were introduced to support generalization, aggregation, qualified association, constraints and a special notation to represent link attributes which correspond to the Structured Analysis concept of "associative entities".

The object model is produced in eight steps, which are applied iteratively:

1. Identify object classes.
2. Prepare a data dictionary.
3. Identify associations.
4. Identify attributes of objects and links.
5. Organize and simplify object classes using inheritance.
6. Verify that access paths exist for likely queries.
7. Iterate and refine the model.
8. Group classes into modules.

**Identify object classes.** Objects and classes are normally described in the user requirements as nouns. In the first stage, accept all the nouns as candidate object classes and, in the next stage, discard unnecessary or incorrect ones and, if necessary, add others which were ignored in the requirements, but which we have now found to be important.

For the banking system, the initial list of candidate object classes is shown in Table 2.1.

| Candidates |
|---|
| Client |
| Money |
| Account |
| Current balance |
| Bank |
| Automatic teller |
| Cheque |
| Transfer |
| Other bank |
| Card |
| Standing order |
| Cash |
| Counter teller |
| Code |
| Savings account |
| Cheque account |

Table 2.1: Candidates for objects

Some of the listed candidate object classes are only attributes of other object classes. Examples are `Current balance`, `Cash` and `Money` (`Cash` and `Money` represent the same concept), which belong to the object class `Account`, and `Code` which belongs to the object class `Card`. The candidate for object class `Bank` is the system we want to analyse, and therefore should be ignored. The candidate for object class `Transfer` is really an operation. The final list of object classes is as depicted in Table 2.2.

| Client |
|---|
| Account |
| Automatic teller |
| Cheque |
| Other bank |
| Card |
| Standing order |
| Counter teller |
| Savings account |
| Cheque account |

Table 2.2: Final list of objects

**Prepare a data dictionary.** Names (nouns) are not enough to describe a concept (an object class, in this case). In order to give a better idea of each object class we write one paragraph for each one, giving any information we think is relevant.

**Account:** allows transactions on it. A single account can have more than one "holder". An account can be of two types: savings and cheque.

**Automatic teller:** a place where the clients can make their own transactions, using cards as identification. We are only interested in the interface between an automatic teller machine and the system where all the information is kept.

**Card:** is assigned to a client and refers to an account. It is used to operate the automatic teller machine. Each card contains a card number and a bank code. Each card is owned by a single client and one client can have more than one card.

**Cheque:** is used by the client to withdraw cash from their own account directly in the bank or is used to credit another client's account and withdraw from the cheque owner's account. The cheque owner may or may not be a client of our bank.

**Cheque account:** is one type of account which has cheques associated with it and can be accessed by an automatic teller machine.

**Client:** holds one or more accounts in the bank and for each account can have zero or one card (optional).

**Counter teller:** represents the interface the bank employee uses to access the accounts.

**Other bank:** interacts with the system, by sending and receiving information about transfers to or from other banks.

**Savings account:** is one type of account which can pay interest. Transactions in this account can imply a loss of interest.

**Standing order:** is a client authorisation to do periodical withdrawals from the client account and respective deposits in somebody else's account which can either be, or not be, in our bank. A standing order is defined for two accounts, one date and one amount.

**Identify associations.** An association is a dependency between classes. Associations are often described in the user requirements as verbs or verb phrases (be careful because services are also described as verbs). At this stage we should not spend too much time deciding about the type of association (aggregation and generalization are a kind of association). Some of the associations may not be in the requirements. Do not forget that the initial requirements are not an immutable document. Analysis is the process of understanding the requirements and proposing a model

for it. The analysis process is iterative, therefore things can change. The first attempt to find associations is depicted in Table 2.3.

| |
|---|
| clients can take money from their own accounts |
| (clients) ... using automatic tellers |
| withdrawing money from an account can be done by cheque |
| (withdrawing from account) ... by transfer to other banks |
| (withdrawing from account) ... by card |
| (withdrawing from account) ... by card using the teller machine |
| ... account if the client authorizes a standing order |
| depositing money in an account can be done by cheque |
| (depositing money in an account) ... by transfer from other banks |
| (depositing money in an account) ... by giving actual cash to the counter tellers |
| in order to access an automatic teller ... (card) |
| the client has a key card |
| the teller machine allows the client |
| |
| *Knowledge of the problem domain* |
| key card accesses cheque accounts |
| bank employs counter tellers |
| client writes cheque |
| counter teller initiates transaction with cheque |
| savings account is an account |
| cheque account is an account |

Table 2.3: Candidates for Associations

The candidate associations are parts of the sentences of the initial set of requirements. In the above list, for example, the entry

```
... account if the client authorizes a standing order
```

incorporates the implicit association

```
Standing order with respect to Account
```

Due to the way used to find associations (seek for verbs and verb phrases), many of the entries listed above are written as actions. Actions represent services, not associations. We will later decide that some of the entries are services. The others we need to rephrase.

The candidate association

```
(clients) ... using automatic tellers
```

describes part of the interaction cycle between the client and the teller machine, not a relationship between the client and teller machine. The same happens with the entries:

```
(withdrawing from account) ... by card using the teller machine
in order to access an automatic teller ... (card)
the teller machine allows the client
```

The entry

```
bank employs counter tellers
```

is an association outside our system, since we are not interested in modelling the staff of the bank.

Notice that we ignored any associations between classes that were eliminated in the previous step. In OMT those associations can be listed and then ignored. We think that this is an unnecessary step which can be avoided.

The final list of associations is shown in Table 2.4. During the construction of the preliminary object model, other associations can be found.

We then look for multiplicity. (Be careful with redundant associations. Study the cardinality in order to decide if an association is really needed — this study can add new associations.)

Figure 2.1 represents a first approach to the object model.

**Identify attributes of objects and links.** A link is an instance of an association. Attributes describe properties of the objects. Attributes are usually described as nouns followed by possessive

| Client owns Account |
| --- |
| Cheque updates Account |
| Other Bank updates Account |
| Card accesses Account |
| Automatic Teller updates Account |
| Client authorizes Standing Order |
| Account is updated by Standing Order |
| Counter Teller initiates transaction with Cheque |
| Counter Teller updates Account |
| Client has Card |
| Client writes Cheque |
| Savings Account is an Account |
| Cheque Account is an Account |

Table 2.4: Final list of associations

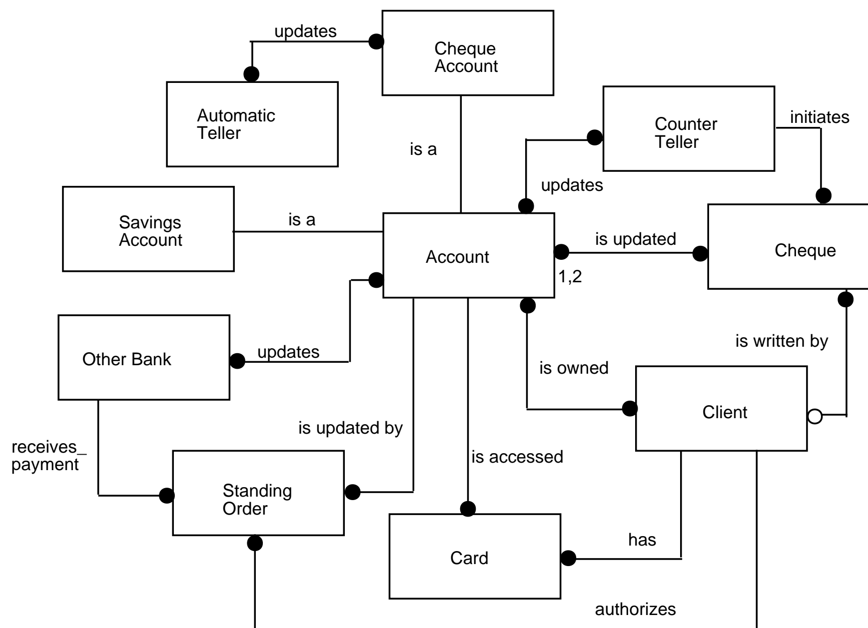



Figure 2.1: Initial approach to an object model diagram

phrases (for example, the name of the client) or by adjectives (for example, expired (date)). Attributes are not fully described in the initial set of requirements. In order to find them we should apply our knowledge of the system (in the real world). Implementation attributes should be avoided at this stage, only important attributes are kept in the object model. Link attributes are usually identified at this point. Figure 2.2 shows the object model with attributes incorporated.

**Organize and simplify object classes using inheritance.** Inheritance can be added in two directions: bottom-up (generalizing common aspects of existing class and creating a superclass) and top-down (refining existing classes into specialized subclasses).

In our example, `Cheque Accounts` and `Savings Accounts` are subclasses of the superclass `Account`.

Sometimes, when the same association name appears more than once with exactly the same meaning, that suggests that the involved object classes can be generalized (bottom-up) by creating a superclass. In our example `Automatic Teller` and `Counter Teller` are subclasses of the new superclass `Entry Station`. `Other Bank` and `Cheque` could, at first sight, be mistaken as subclasses of `Entry Station`, since both are the source of transactions to update an account. However, the services they provide are very different from the ones provided by the tellers. When superclasses

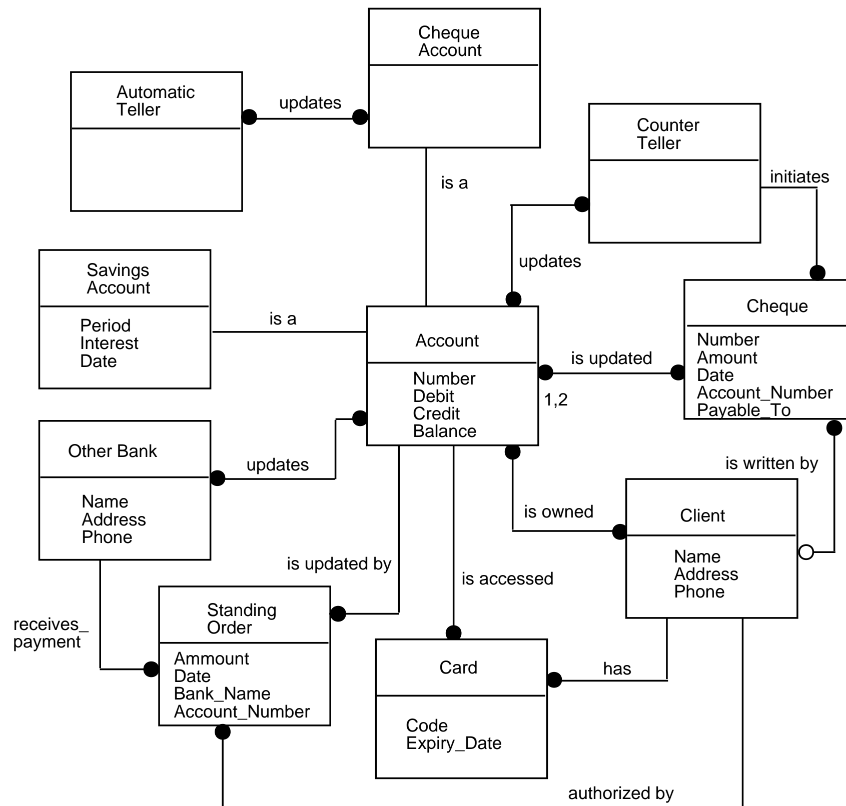



Figure 2.2: Object model with attributes

are identified, attributes and associations can suffer some changes in order to be added to the right object class in the class hierarchy (see Figure 2.3).

**Verify that access paths exist for likely queries.** Now we should examine the object model presented in Figure 2.3 and identify if there are any missing associations which are required to respond to any question we would like the object model to answer. With this study, we can find some missing information. By analysing the association `initiates` between `Counter Teller` and `Cheque` a question arises: is that an association or a flow of control? We know that a cheque transaction will be initiated by a counter teller, but we have not yet decided how autonomous `Cheque` is going to be. In cases where we are not completely sure, we just leave it to be studied later, when our knowledge about the problem has increased.

As `Standing Order` is created and cancelled by `Counter Teller` another association `initiates` can be added to the model in Figure 2.3.

For any question we would like the system to answer it is necessary to evaluate whether or not the necessary associations in the object model exist.

Some decisions are hard to make during the analysis phase. Whether an association is a relationship that is eventually going to be implemented as a pointer attribute, or if it only represents some kind of visibility between objects, or any other kind of interaction, should not be taken too seriously at this stage.

Many-to-many associations do not allow the single identification of the objects involved. We can leave this kind of problem to be dealt with in the design phase.

**Iterate and refine the model.** Usually, an object model needs various iterations until the final version is found.

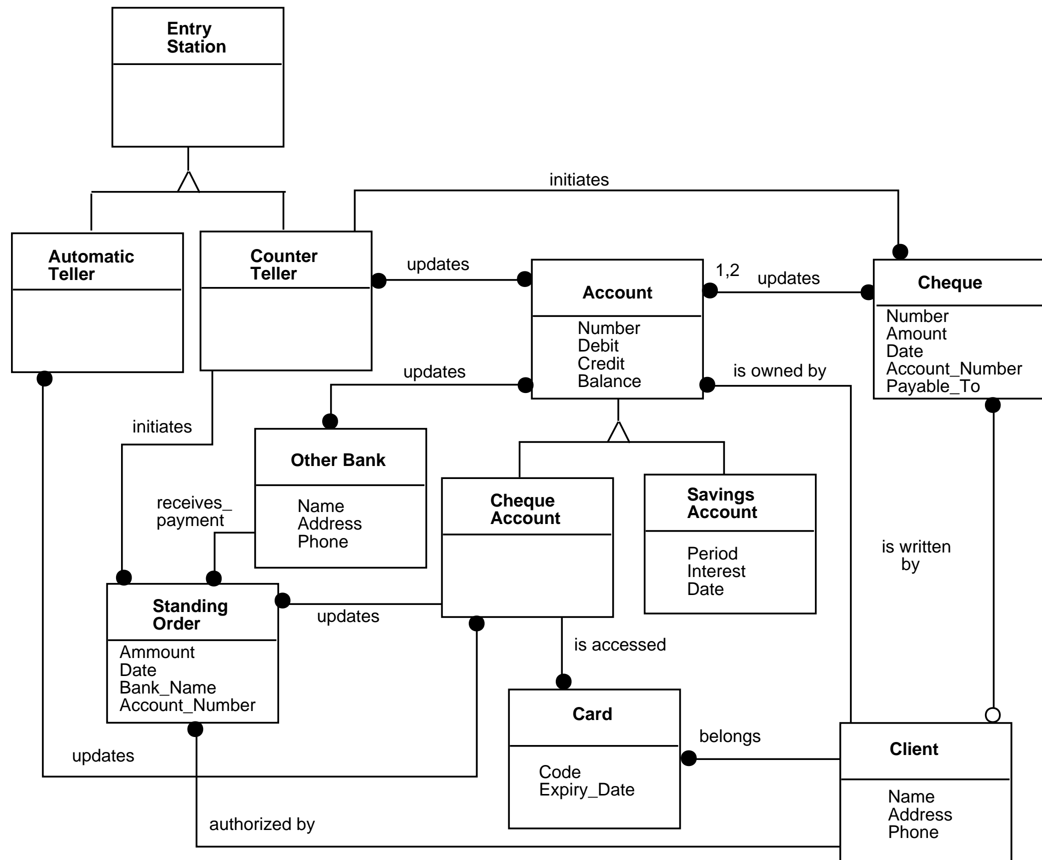



Figure 2.3: Object model with inheritance

**Group classes into modules.** In our example, as we are only considering part of the real problem, we do not really need to split the system into modules. However, it could be done. The modules might be: external interfaces (counter tellers, automatic tellers, other banks) and accounts (account, card and client) and complex operations (standing order and cheques).

### 2.2.2 The dynamic model

For this exercise, we will build the dynamic and functional models. The dynamic model gives the dynamic structure of the system, by describing the dynamic behaviour of each object and the interactions among objects. OMT proposes state diagrams to represent the dynamic model. They are created in five steps:

1. Prepare a scenario.
2. Draw interface formats.
3. Identify events between objects.
4. Build a state diagram.
5. Match events between objects.

**Prepare a scenario.** A scenario gives typical dialogs between the outside world and the system. This help us to understand the way the system should behave. Instead of trying to create the

| |
|---|
| AT offers services to the user (withdrawal, print mini statement); |
| user selects withdrawal and introduces the amount of cash. |
| AT verifies the user balance with the account object; |
| updates account and receives from it the new balance; |
| AT gives the money and prints a receipt. |
| AT offers services to the user (withdrawal, print mini statement); |

Table 2.5: Automatic teller machine scenario

complete dynamic model of the system in one step, we should approach that model by writing scenarios. Table 2.5 is an example of a scenario with the automatic teller interface (AT).

The services offered by a counter teller interface (CT) can be more complex than the ones offered by automatic tellers, involving more object classes. Examples are the case of a cheque deposit (see Table 2.6) and a transfer to another bank (see Table 2.7).

| |
|---|
| CT offers to the bank employee services (withdraw cash, deposit cash, withdraw cheque, |
| deposit cheque, give balance, open an account, close an account); |
| employee selects deposit cheque. |
| CT accepts cheque and sends the its information to object cheque; |
| object cheque identifies the cheque (cheque from this bank or from another bank); |
| cheque belongs to another bank; |
| cheque object withdraws the cheque from the right account and credits the payable account. |
| CT gives a receipt. |
| CT offers services to the bank employee. |

Table 2.6: Deposit cheque scenario

| |
|---|
| CT offers to the bank employee services (withdraw cash, deposit cash, withdraw cheque, |
| deposit cheque, give balance, transfer money, open an account, close an account); |
| employee selects transfer money. |
| CT accepts the information about transfer and checks if client account has enough credit; |
| account has credit; correct amount is debited in object account; |
| sends transfer to the object other bank. |
| CT gives a receipt. |
| CT offers services to the bank employee. |

Table 2.7: Transfer service scenario

Some scenarios will handle special cases (errors and exceptions). In general, we feel that this part can be long, but should be added, to show complex sequences of operations about which we need a more detailed study.

**Interface formats.** Interactions can usually be separated into a user interface and a logic application. In the analysis phase we should concentrate on the flow of information and control, rather than on the presentation format. However, for some applications, user interfaces are very important. It is known that object-oriented programming languages are well suited to the design of human interaction interfaces.

In our example we will not define the interface formats.

**Identify events between objects.** The scenarios help us identify events. Events include signals, inputs, decisions, interrupts, etc. from the users or external devices. The event trace diagram in Figure 2.4 summarizes some of the events between the automatic teller class and all the other classes involved.

Each vertical line in the event trace diagram represents an object and the horizontal arrowed lines are messages passed between processes. For a given process, input messages are events and output messages are events sent to another process.

Some of the external counter teller events lead to a event trace diagram similar to the one presented for automatic teller, but some others interact with other object classes, such as when a

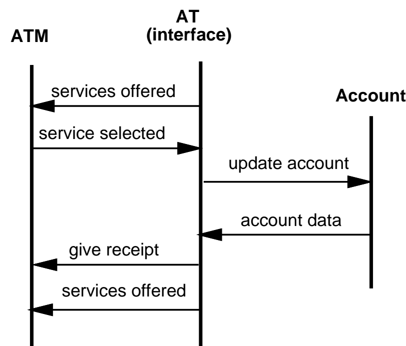



Figure 2.4: Event trace diagram for automatic teller

cheque is credited. The event trace diagram in Figure 2.5 shows the relevant events when a cheque deposit is initialized by a counter teller.

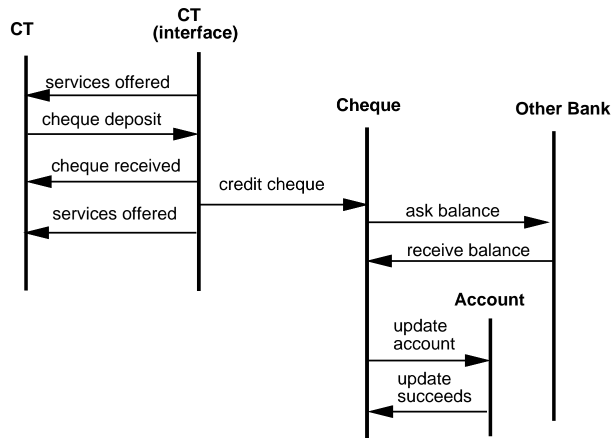



Figure 2.5: Event trace diagram for deposit cheque initiated by CT

Another external event which involves other object classes is a transfer to another bank, also initiated by counter teller (see Figure 2.6).

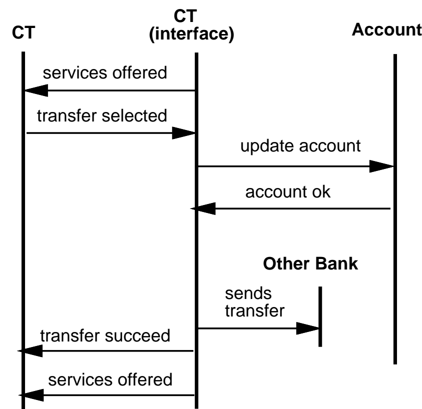



Figure 2.6: Event trace diagram for transfer initiated by CT

Error conditions can be incorporated into the event trace diagram. In this case, the diagram is broken into another diagram which completes the first. We decided to use these diagrams to represent the "optimistic" view and deal with errors in the state diagrams.

An alternative is to represent these events and the object classes involved, by drawing an event flow diagram, such as the one shown in Figure 2.7. It does not give the order in which the events occur. This diagram gives less information than a event trace diagram and so we only use the latter.

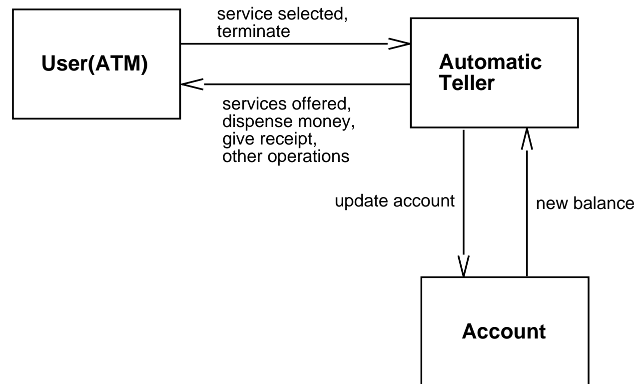



Figure 2.7: Event flow diagram for automatic teller

**Build a state diagram.** State diagrams represent the dynamic behaviour of each object class. They are built from the information given in the event trace diagrams. For a given process, an input message is an event represented as a transition from one state to another, and an output message corresponds to an action inside a state which usually causes an event to be sent to another process.

Automatic tellers have only two initial external events, a withdrawal request or a print balance and mini statement request. A general state diagram for an automatic teller is depicted in Figure 2.8.

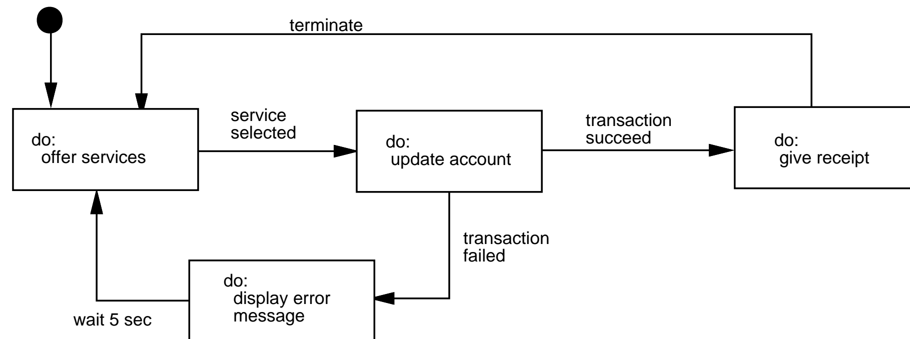



Figure 2.8: State diagram for automatic teller

Time is introduced here as a transition, to mean that an error will be displayed for 5 seconds, before it is erased.

By analysing the external events suffered by a counter teller we realize that they all produce similar event trace diagrams, even if they relate to different objects. Hence, a general state diagram can be produced (see Figure 2.9). Of the external events that arrive at the counter teller, deposit cheque affects it differently. Because of this we introduce a special path in the diagram. Notice that at this stage we do not know whether or not a `cheque transaction` will be successful.

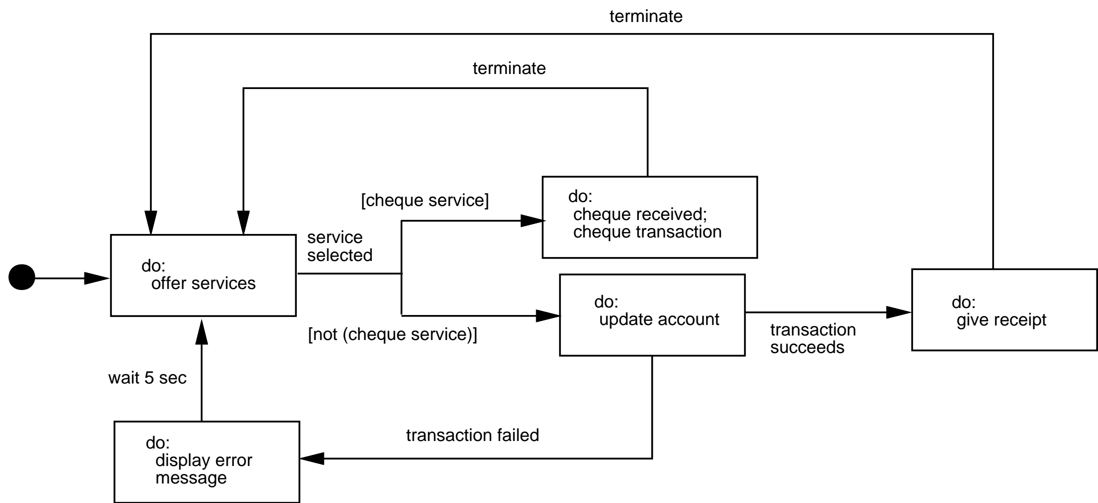



Figure 2.9: State diagram for counter teller

Another process that needs to be analysed is `Cheque`. We can identify some events in the event trace diagrams in Figures 2.5 and 2.6. With the knowledge we already have of the system, we can identify other events and resulting states. The high level state diagram is represented in Figure 2.10. The activity `cheque transaction` in the diagram in Figure 2.9 can be seen as the event that wakes up the process `Cheque`.

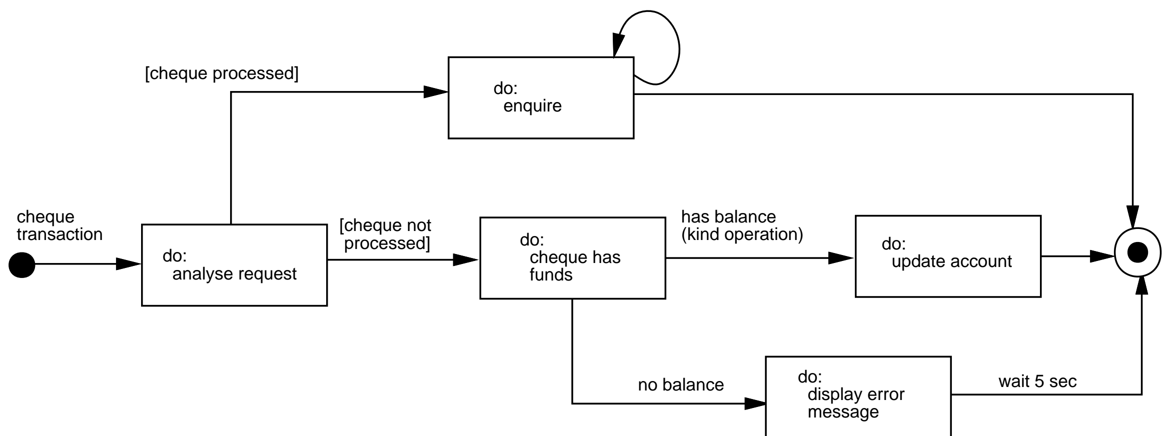



Figure 2.10: State diagram for cheque

We know that a withdraw or a deposit can only happen once in a cheque life time, but a query to the state information can happen any time; this is shown in Figure 2.10. A nested state diagram is used to decompose the state `cheque has funds` (see Figure 2.11).

It is not necessary to draw state diagrams for all the objects in the system. Some objects, such as account, client and card, are passive and their response to events does not affect control. A list of the input events for each of these objects and the output events sent in response to each input event is enough.

**Match events between objects.** The goal of this step is to verify consistency between the state diagrams. In our example, an account can be accessed concurrently by more than one external device (automatic teller, counter teller and other bank). Access to each account should therefore be controlled to ensure that no more than one operation at a time is allowed.

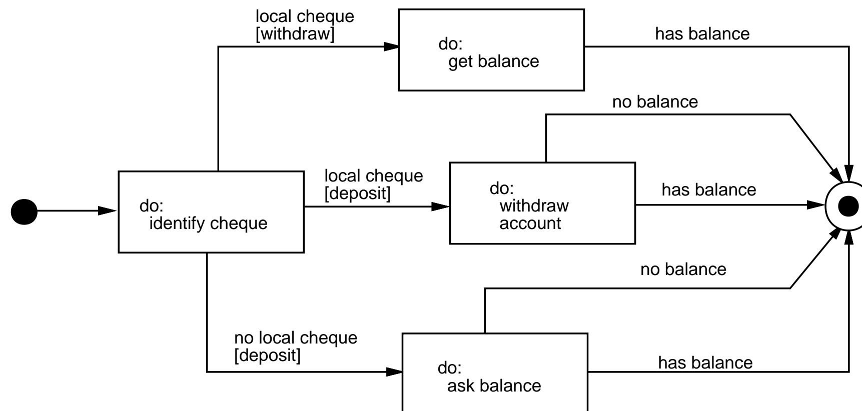



Figure 2.11: State diagram for `cheque has funds`

The set of state diagrams produced for object classes with important dynamic behaviour constitute the dynamic model of the system.

### 2.2.3 The functional model

The functional model represents the computations in the system, i.e., shows how an input value is transformed into an output value. DFDs are useful in representing a functional view which ignores sequence or decision. A DFD process corresponds to activities or actions in the state diagrams. A DFD data flow correspond to objects or attribute values in the object diagram.

The functional model is created in five steps:

1. Identify input and output values.
2. Build data flow diagrams.
3. Describe functions.
4. Identify constraints between objects.
5. Specify optimization criteria.

**Identify input and output values.** Input and output values can be identified in the state diagrams. They are parameters of events between the system and the outside world. Some events in the state diagrams do not suply any data, such as `terminate` and `wait 5 sec`.

Figure 2.12 shows the input and output values between the system and the external devices (automatic tellers, counter tellers, other banks).

**Build data flow diagrams.** DFDs can now be constructed, showing how each output value is produced from input values. Figure 2.12 can be seen as the context diagram of banking system, where `Automated Banking Application` is our system and `Counter Tellers (employee)`, `Automatic Teller Machines` and `Other Banks` are the external entities with which the system communicates. By expanding the process `Automated Banking Application`, we built the DFD in Figure 2.13. (The notation followed is the one proposed by Gane and Sarson [7].)

The diagrams in Figure 2.14 and 2.15 show the processes `accept transaction` and `perform transaction`, respectively.

The diagram in Figure 2.16 shows a detailed data flow diagram for the process `deposit cheque` given in Figure 2.15. The data stores `Account` and `Cheque` are repeated only to improve readability.

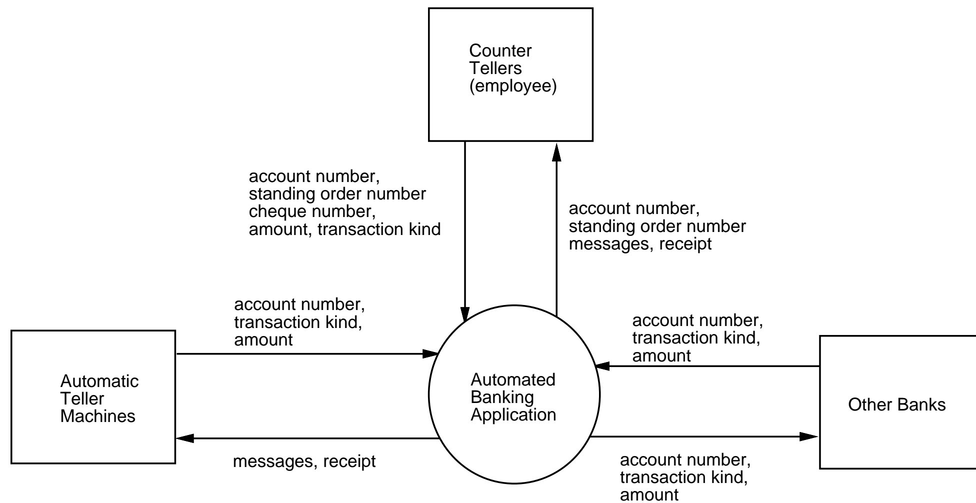



Figure 2.12: Input and output values for the automated banking application

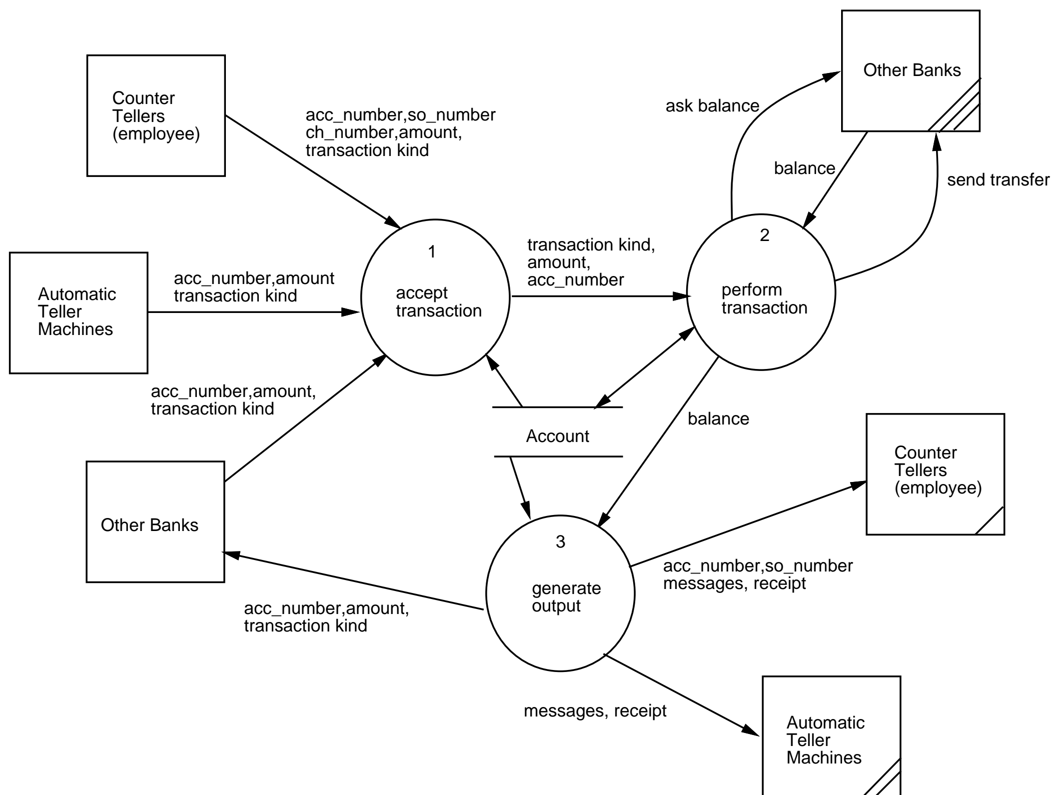



Figure 2.13: Data flow diagram for automated banking application

The problem we are analysing does not have many computations and those it does have are not complicated. For other kind of problems, the functional model (in particular the DFD) can play a more important role.

**Describe functions.** The description of each function helps us to specify what each low level process does. This can be done by using structured English, pseudocode, or a declarative descrip-

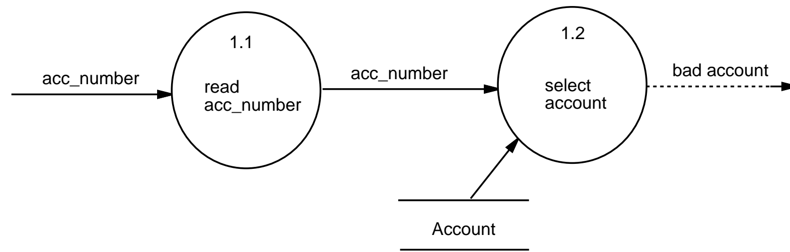



Figure 2.14: Data flow diagram for `accept transaction`

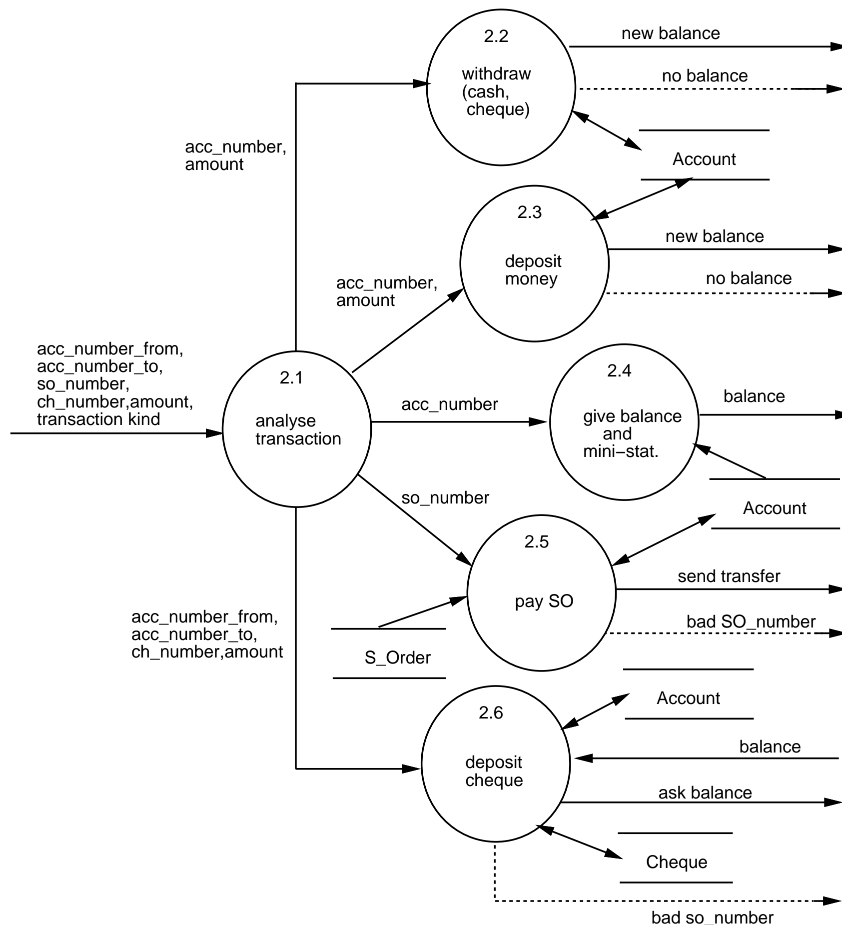



Figure 2.15: Data flow diagram for `perform transaction`

tion which specifies the relationships between input and output values and the relationships among the output values. Depending upon the detail we already have in the DFDs, this description can be more or less complicated. In Table 2.8 we show a description for the process `withdraw (cash or cheque)`, by using pseudocode. A similar description could be done for each process in the data flow diagrams.

Depending upon the kind of problem studied, some techniques are better suited than others. In this example, we did not gain much advantage from using DFDs. In fact, we do not believe that DFDs can really be helpful when an object-oriented approach is required. A DFD seems to go against the fundamental idea of object-oriented thinking: operations are encapsulated by objects. We can draw a diagram in a way that each process would only deal with one data store, but that

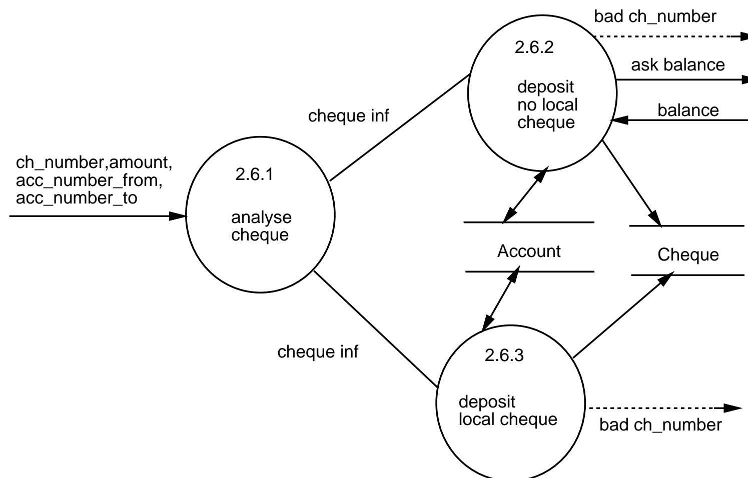



Figure 2.16: Data flow diagram for `deposit cheque`

| withdraw cash and cheque (acc number, transaction kind, amount) $->$ new balance, message |
|---|
| If amount > current balance, |
| reject transaction |
| If amount <= current balance, |
| debit the account |
| If transaction = withdraw cheque, |
| create cheque |

Table 2.8: Function description for `withdraw (cash or cheque)`

will not be useful for complex systems.

For other kind of problems DFDs are very helpful, but we do not believe that in those cases the object oriented approaches should be used.

**Identify constraints between objects.** One constraint we could add to our system is about whether or not a balance could be negative. If accounts with overdraft privileges were accepted, more information should be added to the dynamic and functional models.

**Specify optimization criteria.** Optimizations, are usually difficult to specify completely at this stage. However, when possible, it should be done. In our example, we could propose as optimizations: minimize the number of physical messages between different sites and minimize the time an account is locked for concurrency reasons.

### Adding operations

In OMT, operations are given less emphasis than in other object-oriented methods, such as Coad and Yourdon [5]. According to Rumbaugh, the list of operations we can define for each object is open-ended and it is difficult to decide when to stop adding them. Rumbaugh refers to three different kinds of operation, corresponding to queries about attributes or associations (selectors) in the object model, to events in the dynamic model and to functions in the functional model. These operations are discovered during the following steps:

- Operations from the Object Model.
- Operations from Events.
- Operations from State Actions and Activities.

- Operations from Functions.
- Shopping List Operations.
- Simplifying Operations.

**Operations from the object model.** Rumbaugh proposes that operations to read and write attribute values and association links need not be made explicit in the object model. We think that updating operations should be explicitly added to the object model. During the analysis it is assumed that all attributes are visible. A “dot” notation can be used to indicate an attribute access, such as “Account.balance”.

**Operations from events.** Each event sent to an object corresponds to an operation on that object. Rumbaugh advocates that, during the analysis, events are better represented as labels in the transitions in the state diagrams than as operations in the object model.

**Operations from state actions and activities.** Actions and activities in the state model can be functions. If these functions are important, because they include interesting computations, they should be represented as operations on the object model.

**Operations from functions.** Each function in a DFD corresponds to an operation on an object or several objects (since, as we discussed, a process in a DFD does not directly correspond to an operation in an object). These functions usually have interesting computations, and so should be included in the object model. In Figure 2.15 the functions `withdraw (cash, cheque)`, `deposit money`, `give balance and mini statement` should be considered as operations in object `Account`. The function `pay SO` belongs to the object `Standing Order` and the functions `deposit local cheque` and `deposit no local cheque` are added to the object `Cheque`.

Sometimes we include functions in the DFD that only traverse the object model, without doing any computation or giving any relevant information. This is the case with `select account` in Figure 2.14 which should be omitted in the object model.

If the same sequence of pseudocode fragments appears in different functions, then they can be grouped into a new operation to simplify the functional model.

**Shopping list operations.** “Shopping list” operations are defined by Meyer [10] as the operations that do not depend on a particular application, but are meaningful in their own right. Shopping list operations allow us to consider future possible needs. They provide a way of going beyond the narrow needs of a particular problem (we can see it as a way to incorporate reusability).

In our example, some of these operations are:

```
account :: close
counter-teller :: create-savings-account(customer) -> account
counter-teller :: create-cheque-account(customer) -> account
```

**Simplifying operations.** Use inheritance where possible to reduce the number of distinct operations, by introducing new superclasses as needed. But, be careful to not introduce inheritance for the sake of it. We must avoid unnatural or forced superclass/subclass structures. Whenever a new superclass is introduced it is necessary to locate operations in the right level of the class hierarchy.

Our example is not complex enough to require this step.

#### Iterating Analysis

We should never consider our first analysis model as the final one. As we proceed with the problem, our understanding increases and we then can iterate the analysis steps in order to refine the models already built.

Rumbaugh propose three steps for refining analysis:

- Refining the Analysis Model.
- Restating the Requirements.
- Analysis and Design.

**Refining the analysis model.** It can happen that parts of the model do not fit well. Try to fix it, by analysing each particular problem in more detail. Changes are easier to do early than late, and so they should not be put off.

Sometimes a physical object can play more than one role in the problem. Split it, by creating an object for each role played. This is the case with `Cheque` as we show in the next section.

Check the associations to see if they look right. If one appears strange, analyse it and decide whether to keep it or not.

**Restating the requirements.** After constructing our object model, we should go back and analyse the requirements. It is possible that some of the requirements specify performance criteria; those should be stated in the optimization step. Other requirements can be considered solution proposals; these should be separated and challenged, whenever possible.

The final model should be verified with the client (or user of the system). During the analysis we can find requirements which are incorrect, incomplete or missing. These should be corrected and discussed with the client. It is a good idea to give the model to people who know about the real world problem for verification. The final model should be easy to understand for non-expert computer people.

The final analysis model is a basis for the system design and implementation.

The original set of requirements should be revised to incorporate information discovered during the analysis.

**Analysis and design.** The aim of analysis is to specify the problem domain without introducing implementation dependencies, but they can be very difficult to avoid completely.

The frontier between the different phases in the software life cycle are still difficult to define.

## 2.3 Applying ROOA

### Task 1: Build an object model

Already done, by applying the OMT method.

### Task 2: Refine the object model

Our goal is to have an object model which includes:

- interface objects;
- class templates with services and attributes;
- conceptual relationships between objects and
- messages connections between objects.

We can collect all information from the OMT models in an OCT, as showed in Table 2.9. Only a few of the services shown in column 2 were identified during the construction of the OMT dynamic model, in particular in the event trace diagrams and the state charts. However, as we discussed before, the OMT method does not emphasise services during the analysis phase, as we do. Therefore, we have used the ROOA guidelines to identify services (and attributes). Also, OMT does not deal with message connections in the object model. We see this as a problem, as we may end up drawing associations in the object model which are in fact message connections. To help us with this problem, we can use the event trace diagrams to identify message passing between objects.

| **Class Templates** | **Offered Services** | **Required Services** | **Clients** |
|---|---|---|---|
| Entry_Station (ES) | withdraw_cash | Account.withdraw | External |
| Counter_Teller (CT) | open_account | Account.create | External |
| | close_account | Account.remove | External |
| | deposit_cash | Account.deposit | External |
| | give_balance | Account.balance | External |
| | deposit_cheque | Cheque.deposit | External |
| | ask_transfer | Account.withdraw | External |
| | | Account.deposit | |
| | | OB.send_transfer | |
| | set_standing_order | SO.create | External |
| | cancel_standing_order | SO.cancel | External |
| Automatic_Teller (AT) | mini_statement | Cheque_Account.print_mini_stat | External |
| Other_Bank (OB) | receive_transfer | Account.deposit | External |
| | send_transfer | | CT, SO |
| | cheque_withdraw | | Cheque |
| | remote_withdraw | Account.withdraw | External |
| 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 | |
| | | Account.perhaps_deposit | |
| | | OB.cheque_withdraw | |
| | | Account.full_deposit | |
| Account (A) | create | | CT |
| | remove | | CT |
| | deposit | | CT, Cheque, OB, SO |
| | withdraw | | CT, ES, Cheque, OB, SO |
| | balance | | CT |
| | perhaps_deposit | | Cheque |
| | full_deposit | | Cheque |
| Cheque_Account (CA) | print_mini_stat | | AT |
| Savings_Account (SA) | credit_interest | | internal |
| | update_date | | internal |

Table 2.9: OCT with class templates, services offered, services required and clients

The final object model is shown in Figure 2.17.

## Task 3: Build the LOTOS formal model

LOTOS will give, in a single model, a formal and integrated view of the dynamic behaviour of the system, including a formal and dynamic view of each object class. The static structure of the dynamic model is the same as the object model. Furthermore, it will show the interactions between object classes and the information passed.

To build a LOTOS specification we need to have more detailed information about the services needed. Some of the services were already studied while building the OMT dynamic and functional models. The event trace diagrams are a great help in specifying objects as LOTOS processes.

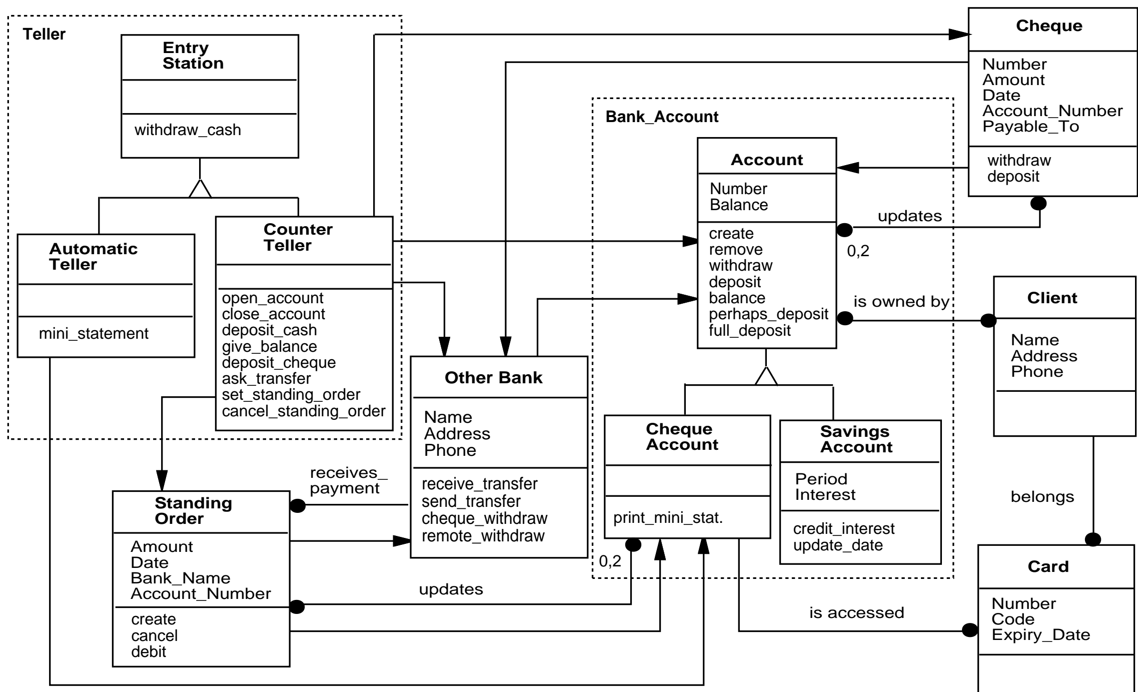



Figure 2.17: Final object model

## Task 3.1: Create an Object Communication Diagram (OCD)

The first step is to complete the OCT (allocate the required gates in the fifth column) so that we can build the OCD. The rules to build an OCD are fully described in [11]. Looking at the second (*Offered Services*) and fourth (*Clients*) columns, we apply two rules to identify the gates:

- Give the same gate name for the object communications which require the same set, or subset, of services; i.e. where there is an overlap between the set of services required by different clients.

- Give different gate names for object communications which require a different set of services, i.e. where there is no overlap between the set of services required by each client.

These two rules are the result of requiring that an object cannot use the same gate to communicate with both an object at the same level of abstraction and another object at a different level of abstraction. (The OMT dynamic and functional models cannot help us on this.) The final OCT is showed in Table 2.10. Based on this, we can build the initial OCD (see Figure 2.18).

## Task 3.2: Specify class templates

The first step before we start writing the LOTOS specification is to decide which object classes [1] are going to be specified as processes plus ADTs and which ones are going to be simple abstract data types.

To start with, we specify each node in the OCD which has connections, i.e. arcs, to other nodes with a process and one or more ADTs, and each node without any arcs as a single ADT. In later refinements we may change this decision, and add processes to the nodes specified as single ADTs.

[1] In this chapter we sometimes use the object-oriented terms used by Rumbaugh *et al.*, and not our own terms. For example, we may use the term *object class* instead of *class template*.

| Class Templates | Offered Services | Required Services | Clients | Gates |
|---|---|---|---|---|
| Teller | open_account(CT) | BA.create | External | t |
| [ES + CT + AT ] | close_account(CT) | BA.remove | External | t |
| | deposit_cash(CT) | BA.deposit | External | t |
| | withdraw_cash(ES) | BA.withdraw | External | t |
| | give_balance(CT) | BA.balance | External | t |
| | deposit_cheque(CT) | Cheque.deposit | External | t |
| | mini_statement(AT) | BA.print_mini_stat | External | t |
| | ask_transfer(CT) | BA.withdraw | External | t |
| | | BA.deposit | | |
| | | OB.send_transfer | | |
| | set_standing_order(CT) | SO.create | External | t |
| | cancel_standing_order(CT) | SO.cancel | External | t |
| Other_Bank (OB) | receive_transfer | BA.deposit | External | ob1 |
| | send_transfer | | Teller(CT), SO | ob2 |
| | cheque_withdraw | | Cheque | ob3 |
| | remote_withdraw | BA.withdraw | External | ob1 |
| Standing_Order | create | | Teller(CT) | so |
| (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 | | |
| | | BA.perhaps_deposit | | |
| | | OB.cheque_withdraw | | |
| | | BA.full_deposit | | |
| Bank_Account (A) | create(A) | | Teller(CT) | ba |
| [A + CA + SA] | remove(A) | | Teller(CT) | ba |
| | deposit(A) | | Teller(CT), | ba |
| | | | Cheque, OB, SO | |
| | withdraw(A) | | Teller(ES,CT), | ba |
| | | | Cheque, OB, SO | |
| | balance(A) | | Teller(CT) | ba |
| | print_mini_stat(CA) | | Teller(AT) | ba |
| | perhaps_deposit(A) | | Cheque | ba |
| | full_deposit(A) | | Cheque | ba |
| | credit_interest(SA) | | internal | |
| | update_date(SA) | | internal | |

Table 2.10: OCT with gates

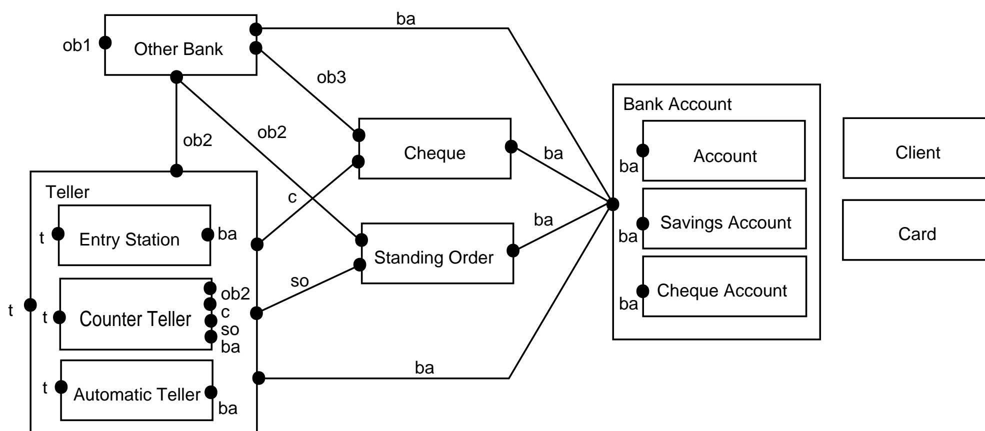



Figure 2.18: Initial OCD

As an example, let us specify the class templates involved in the inheritance hierarchy for accounts and, as an example of a class template which kept changing during Task 3, the class template `Cheque`.

`Account` is an abstract superclass, and so it is modelled as a process with **exit** functionality, as follows:

```
process Account[c](this_account: State_Account): exit(State_Account) :=
  c !deposit !Get_Account_Number(this_account) ?m: Money;
  exit(Credit_Account(this_account, m))
[]
  c !withdraw !Get_Account_Number(this_Account) ?m: Money;
  ( choice if_money: Bool []
    [if_money] -> c !rtn_withdraw !Get_Account_Number(this_account) !true;
                  exit(Debit_Account(this_account, m))
  []
    [not (if_money)] -> c !rtn_withdraw
                          !Get_Account_Number(this_account) !false;
                        exit(this_account)
  )
[]
  c !balance !Get_Account_Number(this_account) !Get_Balance(this_account);
  exit(this_account)
[]
  c !perhaps_deposit !Get_Account_Number(this_account) ?m: Money;
  exit(Credit_Pending(this_account, m))
[]
  c !full_deposit !Get_Account_Number(this_account) ?m: Money ?valid: Bool;
  ( [valid] -> exit(Add_Credit_Pending(this_account, m))
  []
    [not (valid)] -> exit(Sub_Credit_Pending(this_account, m))
  )
endproc
```

the ADT where `Account State` is defined can be modelled 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
      Credit_Pending     : State_Account, Money    -> State_Account
      Add_Credit_Pending : State_Account, Money    -> State_Account
      Sub_Credit_Pending : State_Account, Money    -> State_Account
      Credit_Interest    : 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 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);
        Get_Account_Number(Credit_Pending(a, m)) = Get_Account_Number(a);
        Get_Account_Number(Add_Credit_Pending(a, m)) = Get_Account_Number(a);
        Get_Account_Number(Sub_Credit_Pending(a, m)) = Get_Account_Number(a);
        Get_Account_Number(Credit_Interest(a, m)) = Get_Account_Number(a);
      ofsort Balance
        Get_Balance(a) = Some_Balance;
 endtype
```

where we only define equations for selectors which have to return a particular value; selectors which do not have to return a particular value are specified with a dummy equation, such as the equation for `Get balance`; modifiers are left without equations.

`Account` has two subclasses, `Cheque Account` and `Savings Account`. `Cheque Account` is specified as follows:

```
process Cheque_Account[c](this_account: State_Account) : noexit :=
  ( ( Account[c](this_account)
      >> accept upd_account: State_Account in exit(this_account)
    )
  []
    c !set_mini_statement !Get_Account_Number(this_account) !this_account;
    exit(this_account)
  []
    c !remove !Get_Account_Number(this_account); stop
 ) >> accept upd_account: State_Account in Cheque_Account[c](upd_account)
endproc
```

And Savings Account can be modelled in a similar way:

```
process Savings_Account[c](this_account: State_Account, prd: Period,
                            int_rate: Interest) : noexit :=
  ( hide credit_interest, update_date in
    ( (Account[c](this_account)
        >> accept this_account: State_Account in
           exit(this_account, prd, int_rate)
      )
    []
      credit_interest;
      exit(Credit_Interest(this_account, This_Mon), prd, int_rate)
    []
      update_date;
      exit(this_account, Update_Date(prd), int_rate)
    []
      c !remove !Get_Account_Number(this_account); stop
    )
  ) >> accept upd_account: State_Account, upd_prd: Period, upd_int_rate: Interest
   in Savings_Account[c](upd_account, udp_prd, upd_int_rate)
endproc
```

The three class templates form the subsystem Bank Account. Supposing that we are only dealing with one savings account and one cheque account:

```
process Bank_Account[c] : noexit :=
    ( Savings_Account[c](Make_Account(id1 of Account_Number, Some_Balance),
                         This_Prd, This_Inter)
    |||
      Cheque_Account[c](Make_Account(id2 of Account_Number, Some_Balance))
    )
  where
  process Savings_Account ... endproc
  process Cheque_Account ... endproc
endproc
```

The class template Cheque is modelled by using the two auxiliary processes: Active Cheque and Passive Cheque.

```
process Cheque[c, ob3, ba](ch: Cheque_Number) : noexit :=
  ( Active_Cheque[c, ob3, ba](ch)
   >> accept this_cheque: State_Cheque in Passive_Cheque[c](this_cheque)
  )
endproc
```

Notice that the parameter defined in Cheque is not exactly the kind of sort we would expect. This was the first solution we thought of to deal with cheques. To start with, we only define a

simple ADT which is composed of the attribute Cheque Number. Only then does the template Passive Cheque deal with the full state of cheques. Secondly, notice that we cannot access Passive Cheque until Active Cheque is finished. This sequentiality is introduced by the enable operator >>.

Active Cheque is then specified as follows:

```
process Active_Cheque[c, ob3, ba](ch: Cheque_Number) : exit(State_Cheque) :=
  ( ( c  !deposit ?id: Id_Tellers ?bk: Bank_Name
         ?ch_from: Account_Number ?ch_to: Account_Number !ch ?m: Money;
     ba !perhaps_deposit !ch_to !m;
      ( [bk eq This_Bank] ->
        ( ba !withdraw !ch_from !m;
          ba !rtn_withdraw !ch_from ?cheque_valid: Bool;
          ba !full_deposit !ch_to !m !cheque_valid;
          exit(Make_Cheque(ch, ch_from, ch_to, bk, m))
        )
       []
        [bk ne This_Bank] ->
        ( ob3 !cheque_withdraw !bk !ch_from !ch !m; (* to other banks *)
          ob3 !rtn_cheque_withdraw !bk !ch_from !ch !m ?cheque_valid: Bool;
          ba !full_deposit !ch_to !m !cheque_valid;
          exit(Make_Cheque(ch, ch_from, ch_to, bk, m))
    ) ) )
   []
    ( c !withdraw ?id: Id_Tellers ?bk: Bank_Name
        ?n: Account_Number !ch ?m: Money; ... (* with my cheque in my bank *)
  ) )
 endproc
```

When an instance of Active Cheque asks for the service full deposit from an instance of Cheque Account, the behaviour of Cheque Account depends upon the value of the boolean variable cheque valid. If this variable takes the value 'true', the amount m is added to the current balance, otherwise it is not.

Finally, Passive Cheque can be:

```
process Passive_Cheque[c](this_cheque: State_Cheque) : noexit :=
  ( c !enquire_cheque ?id: Id_Tellers
      !Get_Cheque_Number(this_cheque) !this_cheque;
    exit(this_cheque)
  ) >> accept update_cheque: State_Cheque in
    Passive_Cheque[c](update_cheque)
endproc
```

Having discovered one more service (enquire_cheque), we have to update the OCTs and the refined object model accordingly.

## Task 3.3: Compose the objects into a behaviour expression

During the first iteration of ROOA we deal, in general, with single instances of each class template. (There are situations, as we will see in Chapter 3, where we may want to define object generators in the first iteration.)

After specifying each class template, we can compose them by following the algorithm presented in [3]:

```
( Other_Bank[ob1, ob2, ob3, ba](Make_Bank(...))
 |[ob2, ob3]|
 ( Teller[t, ob2, c, so, ba]
  |[c, so]|
```

```
    ( Cheque[ob3, c, ba](Make_Cheque(...))
    |||
      Standing_Order[ob2, so, ba](Make_SO(...))
   ))
  |[ba]|
   Bank_Account[ba]
  )
 where ...
```

Before we can start the next task, we have to model the interface scenarios and compose them, one at a time, with the behaviour expression above. Here, the scenarios proposed by the OMT method do not help much, as we are only interested in the required services with the respective answers.

Consider the following part of an interface scenario for the banking system:

```
process Interface_Scenario[t, ob1]: noexit :=
  (* create an automatic teller and a counter teller *)
    t !create ?idc: Id_Tellers;
    t !create ?ida: Id_Tellers;
  (* open a cheque account from counter teller *)
    t !open_account !idc !cheque;
    t !rtn_open_account !idc ?nc: Account_Number;
  (* deposit from counter teller *)
    t !deposit_cash !idc !nc !This_Mon;
    t !rtn_deposit_cash !idc !nc !This_Mon;
  (* balance from automatic teller*)
    t !print_mini_statement !ida !nc;
    t !rtn_print_mini_statement !ida !nc ?a: Account;
  (* withdrawal from automatic teller *)
    t !withdraw_cash !ida !nc !This_Mon;
    t !rtn_withdraw_cash !ida ?val: Bool;
    ...
   (hide success in success; stop)
endproc
```

The `Interface Scenario` process acts as if it were the client of the whole system. It initiates calls to the tellers on gate `t` and to other banks on gate `ob1`, and waits for the respective answers. We compose it in parallel with the above behaviour expression, by using gates `t` and `ob1`, as follows:

```
 ( ( Other_Bank[ob1, ob2, ob3, ba](Make_Bank(...))
     ...
   )
  |[ba]|
   Bank_Account[ba]
 )
|[t, ob1]|
 Interface_Scenario[t, ob1]
```

In later iterations, when object generators are introduced to deal with multiple instances, the composed behaviour expression is refined and built from a combination of object generators and 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 the LITE tools to check the syntax and static semantics of the LOTOS specification. Any errors, omissions or inconsistencies found during the simulation will lead us to iterate Tasks 2 and 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

During successive refinements of the LOTOS specification 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

Static relationships can be modelled as attributes and given as parameters of the processes defining the corresponding class templates.

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

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

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

The instantiation of `Cheque Account` is now:

```
Cheque_Account[g](Make_Account(id2 of Account_number, Some_Balance),
                              {} of Card_Number_Set, {} of SO_Number_Set)
```

### Task 3.5.2: Introduce object generators

During a first iteration we deal only with a single instance of each class template. This simplifies the problem and allows us to prototype with a specific number of objects. However, in general, several instances of the same class may be required. This is achieved by defining an *object generator* for a class template.

When dealing with subsystems, we can decide to define an object generator for each component, or else define an object generator for the whole subsystem. Which is to be preferred depends on each particular situation.

ROOA allows variations of object generators. One example, is the one defined for `Cheque Account`:

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

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

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

Because `Cheque Account` and `Savings Account` are subclasses of the abstract class template `Account`, their identifiers have to be of the sort `Account Number` in order to inherit from the same superclass. This is why we divided the identifiers defined in the ADT `Id Type` into several groups (see [11]). Therefore, the ADT `Account_Number_Set_Type` needs to be changed in order to include this information:

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

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

In situations where there is no inheritance involved, the parameter corresponding to `cheque` will not exist.

An object generator for `Cheque` will look different from `Cheque Accounts`. During later refinements of the LOTOS specification, this class template was one of the class templates that suffered more changes. One of the important changes was to consider that we had two kinds of cheque: the ones to withdraw and the ones to deposit. So, we split `Cheque` into a `Cheque_Withdraw` and `Cheque_Deposit`. Another difference in cheques is that the object identifier is not generated by the system. These lead us to define an object generator which can be used to create objects of both kinds:

```
process Cheques[c, ob3, ba] : noexit :=
  c !create_cheque ?ch: Cheque_Number ?bk: Bank_Name
    ?from_acc: Account_Number ?to_acc: Account_Number ?m: Money;
  ( Cheque_Deposit[c, ob3, ba](Make_Cheque(ch, from_acc, to_acc, bk, m))
  |||
    Cheques[c, ob3, ba]
  )
 []
  c !create_cheque ?ch: Cheque_Number ?bk: Bank_Name
    ?n: Account_Number ?m: Money;
  ( Cheque_Withdraw[c, ob3, ba](Make_Cheque(ch, n, n, bk, m))
  |||
    Cheques[c, ob3, ba]
  )
endproc
```

The object generator now has no parameters, but each branch of the choice still has the same structure of the object generators discussed in [11].

### Task 3.5.3: Identify new higher level objects

The identification of new higher level groupings (subsystems and aggregates) leads us to change both the initial OCD and the OCT in order to incorporate the new objects. Therefore we should apply again Tasks 3.1, 3.3 and 3.4.

We grouped `Cheques` with `Standing Orders` to form the subsystem `Financial Instruments`. These two class templates were grouped because they have the same servers and are servers of the same clients. (See the rules given in [11].) These changes can be seen in the OCT represented in Table 2.11.

| Class Templates | Offered Service | Required Service | Clients | Gates |
|---|---|---|---|---|
| Teller | open_account(CT)<br>... | BA.create | External | t |
| Other_Bank (OB) | receive_transfer<br>send_transfer<br>cheque_withdraw<br>remote_withdraw | BA.deposit<br><br><br>BA.withdraw | External<br>Teller(CT), FI(SO)<br>FI(Cheque)<br>External | ob1<br>ob2<br>ob2<br>ob1 |
| Financial_Instrument (FI)<br>[ SO + Cheque ] | create(SO)<br>cancel(SO)<br>debit(SO)<br><br><br>withdraw(Cheque)<br>deposit(Cheque)<br><br><br><br><br>... | <br><br>BA.withdraw<br>BA.deposit<br>OB.send_transfer<br>BA.withdraw<br>BA.withdraw<br>BA.deposit<br>BA.perhaps_deposit<br>OB.cheque_withdraw<br>BA.full_deposit | Teller(CT)<br>Teller(CT)<br>internal<br><br><br>Teller(CT)<br>Teller(CT) | cs<br>cs<br><br><br><br>cs<br>cs |
| Bank_Account (BA) | create<br>remove<br>deposit<br><br>withdraw<br><br>balance<br>... | | Teller(CT)<br>Teller(CT)<br>Teller(AT,CT), OB,<br>FI(Cheque, SO)<br>Teller(ES,CT), OB,<br>FI(Cheque, SO)<br>Teller(CT) | ba<br>ba<br>ba<br><br>ba<br><br>ba |

Table 2.11: Refined OCT

The revised OCD, based on Table 2.10 but after we have introduced object generators, is depicted in Figure 2.19.

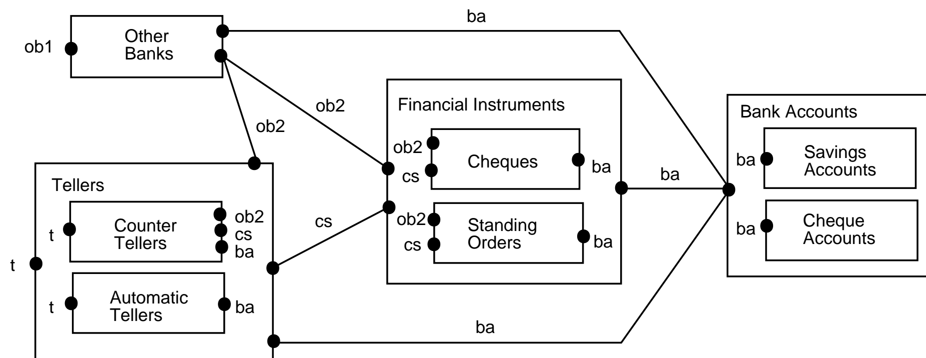



Figure 2.19: Final OCD

Now, the composition of the object generators in the OCD takes the form:

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

#### Task 3.5.4 and 3.5.5: Demote and promote an object to be specified as ADTs or as processes

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.

In this example, `Client` was specified with a single ADT.

#### Task 3.5.6: Refine processes and ADTs

The complete definition of a process or an ADT can be done incrementally. In each refinement we can add more detail to the specification. When more information is added to the formal model, more static relationships, attributes, services, and message connections can be identified. In this case, add them all to the object model and apply again Tasks 2 and 3.

In the specifications we developed using ROOA, there were major differences between the initial ones and the final ones. These differences are of two kinds: structure and detail. The structure of a specification can change according to the extra complex objects and subsystems we find during refinement. The detail involved when specifying each class template for the first time depends on the experience already acquired with ROOA. Even then, we advise complex objects to be specified incrementally. Start by just using the services (and attributes) we understand, leaving for further iterations the ones which we do not understand well. A good example of this was the class template `Cheque`. It kept changing during the application of the method. The final version of the process specifying `Cheque Deposit` is as follows:

```
process Cheque_Deposit[cs, ob2, ba](this_cheque: Cheque) : noexit :=
  ( Cheque_Deposit_1[cs, ob2, ba](this_cheque)
   >> accept this_cheque: Cheque in Passive_Cheque_Deposit[cs](this_cheque)
  )
  where
    process Cheque_Deposit_1[cs, ob2, ba]
              (this_cheque: Cheque) : exit(Cheque) :=
      ba !perhaps_deposit !Get_Acc_To(this_cheque)
         !Get_Amount(this_cheque);
      ( [Get_Bank(this_cheque) eq This_Bank] ->
        ( ba !withdraw !Get_Acc_From(this_cheque) !Get_Amount(this_cheque);
          ba !rtn_withdraw_ok !Get_Acc_From(this_cheque) ?cheque_valid: Bool;
          ba !full_deposit !Get_Acc_To(this_cheque)
             !Get_Amount(this_cheque) !cheque_valid;
          exit(this_cheque)
        )
       []
        [Get_Bank(this_cheque) ne This_Bank] ->
        ( ob2 !cheque_withdraw !Get_Bank(this_cheque)
              !Get_Acc_From(this_cheque) !Get_Cheque_Number(this_cheque)
              !Get_Amount(this_cheque);       (* to other banks *)
          ...
      ) )
```

```
    endproc (* Cheque_Deposit_1 *)

    process Passive_Cheque_Deposit[cs](this_cheque: Cheque) ...  endproc
endproc (* Cheque_Deposit *)
```

With the refinement of the template we had to refine also the ADT `Cheque Type` so that operations such as `Get Amount` were defined.

## 2.4 Conclusions

This chapter discussed the application of ROOA to a simple automated banking system. It started by applying the OMT method and then used the results as an input to the application of ROOA.

The banking system has a simple pattern of communication, but it involves interesting objects such as `Cheque` and it is a good example to discuss inheritance. As we can appreciate from the results, inheritance can be simply modelled in ROOA by using the standard LOTOS constructs. However, this simplicity is only kept when we do not require redefinition of services. When a service needs to be redefined, and in order to guarantee behavioural inheritance, the resulting LOTOS specification is more complex and LOTOS proves itself of little use for this task (see [4] for more details).

At the end of this exercise we could ask ourselves the question: is there any advantage in using a method such as OMT, build all the models and then apply ROOA? The answer to this question is "no". If the models (dynamic and functional) already exist, then we can use them to identify useful information to build the OCT and then the LOTOS specification. However, if those models do not exist yet, it is not worthwhile to build them before applying ROOA. ROOA gives good guidelines to identify services and to start writing a LOTOS specification. And, for a developer with some experience, LOTOS is a better tool to model dynamic and functional behaviour than state diagrams and data flow diagrams.

The LOTOS specification for the automated banking system is given in Appendix A.

# Chapter 3

# Warehouse management system

## 3.1 Introduction

In the first case study, the banking system, the object model dominated. In this second case study, the problem has a much stronger dynamic component. Also, in the first case study, we applied a complete analysis method, so that we would see how ROOA behaved. In this second case study, we started from an object model created by someone else and then we applied ROOA.

The chosen problem is the warehouse management system described by Jacobson [9].

## 3.2 Applying ROOA

### Task 1: Build an object model

The method proposed by Jacobson [9] consists of the construction of the requirements model and the analysis model. Our interest is not to apply the OOSE method, but only to take an object model and use it as a basis to build a LOTOS formal model. Figure 3.1 depicts a simplified version of the case study presented in [9].

The OOSE method includes entity objects, interface objects and control objects. Although these three kinds of object are useful in many situations, in order to fully understand them we need to define them in terms of their services and attributes. Jacobson argues that because the operations change a lot in the design model, it is better not to have them so early in the analysis. We believe that operations (together with attributes) are the way to understand the meaning of an object. A name and a symbol is not enough, especially for control objects. Of course, we have the use cases, but the objects are spread amongst them and it is often difficult to understand why the analyst opted for some of the objects.

In the analysis model of the example given in [9], the necessity for objects such as `Interwarehouse Transporter` and `Local Warehouse Transporter` is difficult to understand. We can identify three reasons for this difficulty: (i) many of the objects identified by Jacobson are graphical user interface objects; (ii) the class templates do not have services (and attributes); (iii) some of the objects identified are in reality actors of the system, i.e. part of the environment, and not objects which we want to model. We solved this problem by removing from the model references to the graphical interface objects and the interactions between external objects. The revised problem is:

> A company has a set of warehouses, distributed throughout the country. Each warehouse has a set of warehouse places where items may be stored. Clients may deliver items to one warehouse and, at a later date, collect them from the same or another warehouse.
>
> When clients wish to enter or remove an item, they contact the office which sends requests to the system to ensure that the item is expected or is available for collection

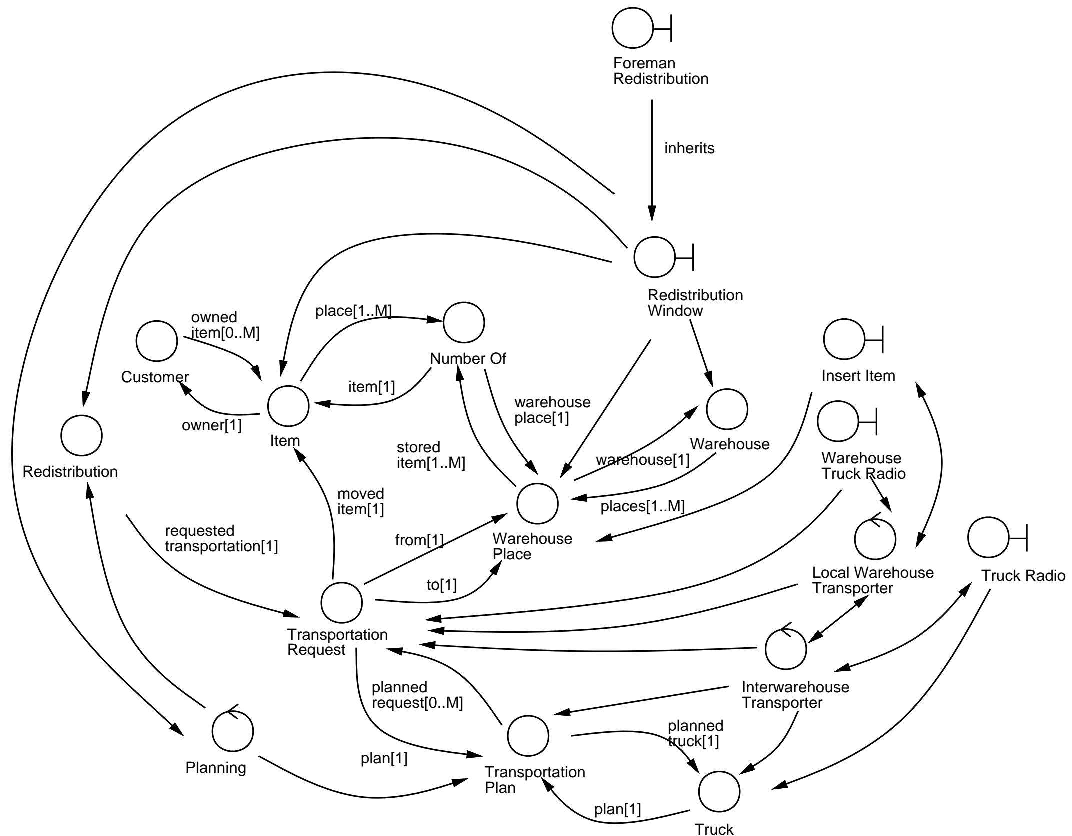



Figure 3.1: Simplified object analysis diagram produced by Jacobson

at the correct place and time. Items may be moved from one warehouse to another if a warehouse is becoming full.

After receiving requests, a computer planning system controls when the requests should be carried out by sending appropriate orders to truck drivers (for inter-warehouse movement) or to forklift drivers to move items between the loading bay and a warehouse place. A local planning system within each warehouse allocates items to warehouse places.

The system should be as decentralized as possible.

Task 2 in ROOA refines the object model created from other OOA methods (Task 1) adding to it:

- interface objects;
- class templates with services and attributes;
- conceptual relationships between objects and
- messages connections between objects.

## Task 2: Refine the object model

This task is composed of three main subtasks: complete the object model, initial identification of dynamic behaviour and structure the object model. To complete the object model, we identify interface objects, add static relationships and add attributes and services. Most of the services

may be identified in the subtask which is concerned with the initial identification of dynamic behaviour. To accomplish this we define interface scenarios, create event trace diagrams and start the construction of the OCT, and add message connections to the object model. Then, we structure the object model, if necessary, by using aggregates, inheritance and subsystems.

Our first problem was to identify attributes and services. For many of the class templates shown in the object model in Figure 3.1, we could not easily identify attributes or services. So, we realised that we did not yet fully understand the object model. The reason was that the warehouse problem is mainly a process-oriented, rather than a data-oriented problem. Therefore, it is much more difficult to build the object model.

In general, to understand a data-oriented problem, we can start drawing object models, but, as process-oriented problems have their focus on dynamic behaviour, that approach does not quite fit. With process-oriented problems, the dynamic aspects of the system must be investigated to help identify the static object structure. We found that event trace diagrams were very useful in discovering what the system is supposed to do.

So, we started drawing event trace diagrams for the main services of the whole system. The more important ones are:

- `initiate_enter new item` which reserves space in a warehouse and creates a request (see Figure 3.2).

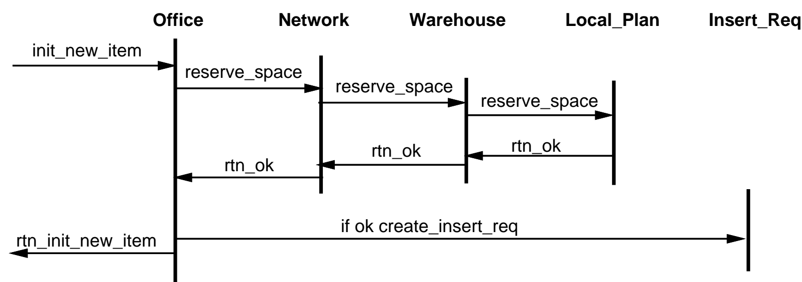



Figure 3.2: Initiate enter new item

- `execute_enter new item` which places a new item in a place in a warehouse (see Figure 3.3).

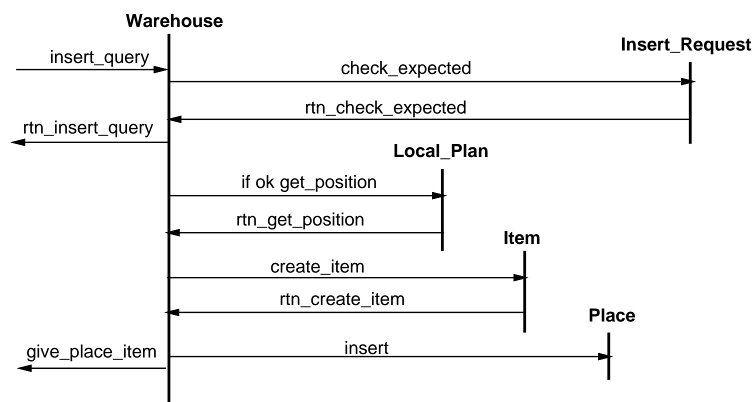



Figure 3.3: Execute enter new item

- `initiate_remove item` which initiates a removal of an item from a warehouse (see Figure 3.4). This can include the initiation of a redistribution request.

- `execute_remove item` which removes an item from a warehouse (see Figure 3.5).

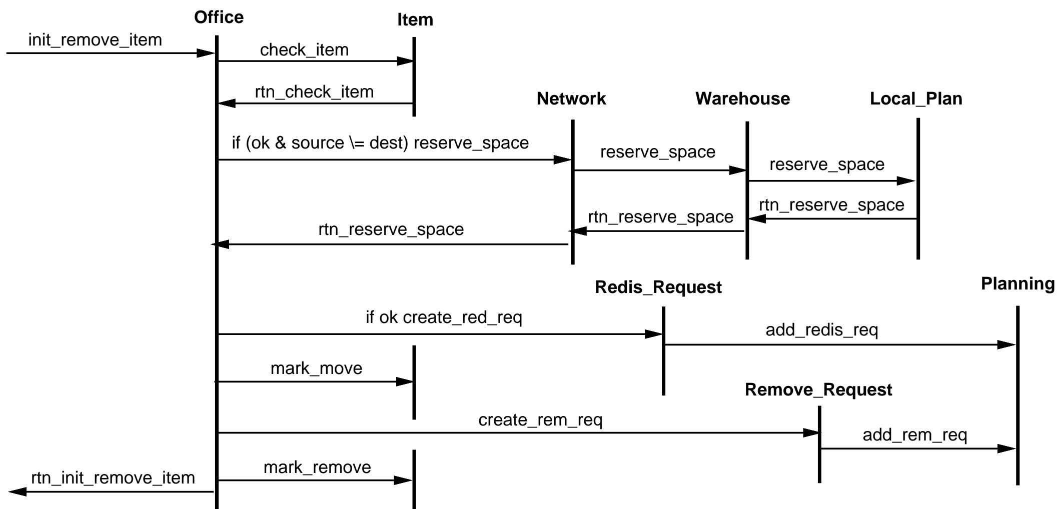



Figure 3.4: Initiate remove item

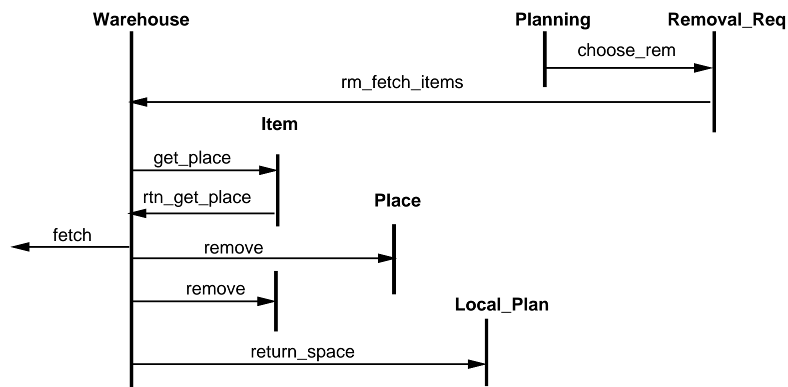



Figure 3.5: Remove item

- `initiate redistribution` which initiates the move of one item from one warehouse to another (see Figure 3.6).

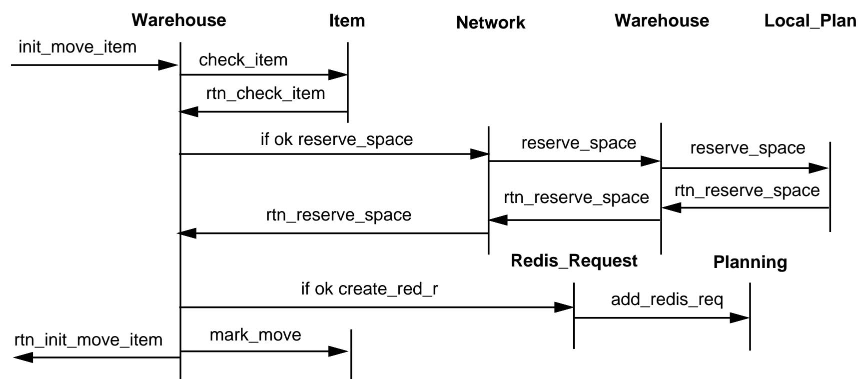



Figure 3.6: Initiate redistribution

- `execute redistribution` which moves an item from one warehouse to another (see Fig-

ure 3.7).

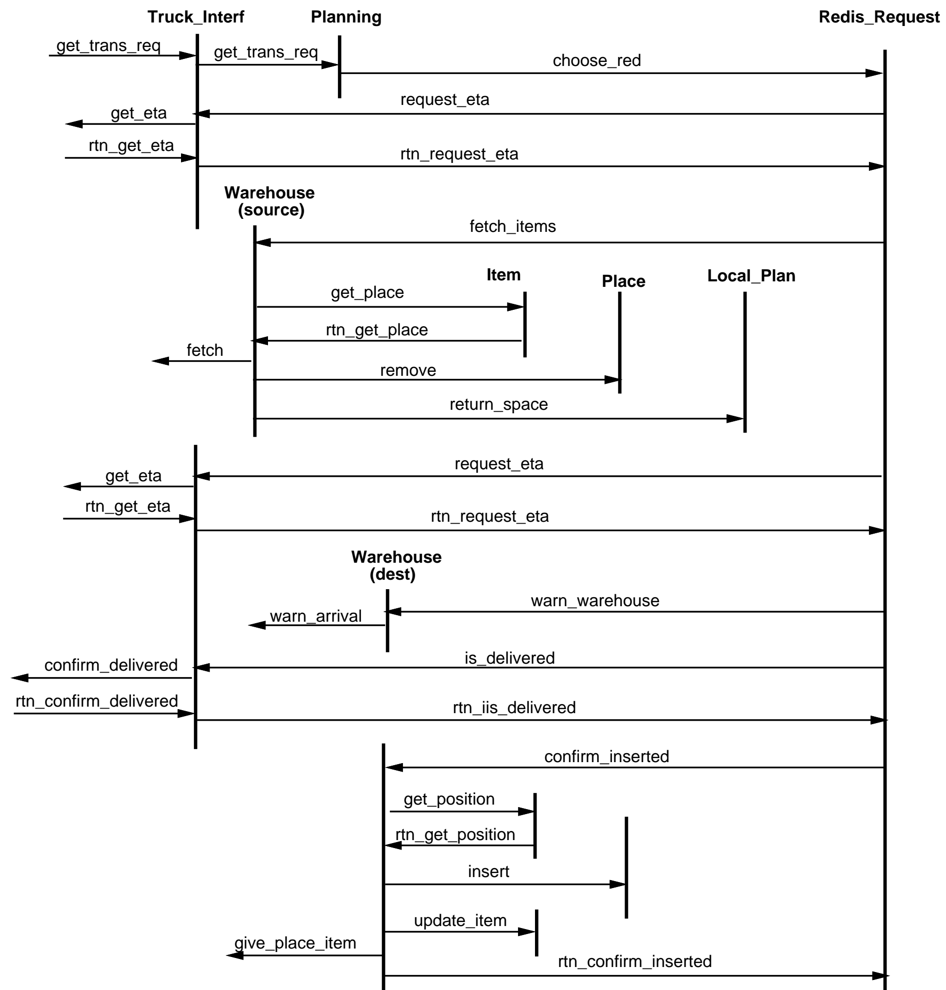



Figure 3.7: Execute redistribution

The information given in the event trace diagrams is collected in an OCT (see Tables 3.1 and 3.2) which is then used to build the final version of the object model. At this point, only the first four columns of the OCT are filled. The fifth column is filled in Task 3.

During this task we also decided that we could group the class templates `Remove Request`, `Redis Request` and `Insert Request` to form a subsystem. The OCT in Table 3.1 shows this grouping.

With our better understanding of the system, and with the information in the OCT we were able to build the object model which is represented in Figure 3.8

## Task 3: Build the LOTOS formal model

### Task 3.1: Create an Object Communication Diagram (OCD)

The first step to be accomplished in this task is to build an object communication diagram (OCD). An OCD is a graph where each arc represents communication between objects and each node represents either an instance of a class template or an object generator.

To be able to build the OCD we have to fill in column five of the OCT, so that we can decide about the gates that are going to be needed. Column 5 of the OCT is filled in according to the

| **Class Templates** | **Offered Service** | **Required Service** | **Clients** | **Gates** |
|---|---|---|---|---|
| Office | init_new_item | Network.reserve_space | External | usr3 |
| | | Request.create_i_r | | |
| | init_remove_item | Item.check_item | External | usr3 |
| | | Network.reserve_space | | |
| | | Request.create_red_r | | |
| | | Item.mark_move | | |
| | | Request.create_rem_r | | |
| | | Item.mark_remove | | |
| Warehouse | init_move_item | Item.check_item | External | usr1 |
| [ + Place + Local_Plan] | | Network.reserve_space | | |
| | | Request.create_red_r | | |
| | | Item.mark_move | | |
| | confirm_inserted | Local_Plan.get_position | Request(Redis_R) | req1 |
| | | Place.insert | | |
| | | Item.update_item | | |
| | | External.give_place_item | | |
| | fetch_items | Item.get_place | Request(Redis_R) | req1 |
| | | External.fetch | | |
| | | Place.remove | | |
| | | Local_Plan.return_space | | |
| | rm_fetch_items | Item.get_place | Request(Rem_R) | req1 |
| | | External.fetch | | |
| | | Place.remove | | |
| | | Item.remove | | |
| | | Local_Plan.return_space | | |
| | insert_query | Request.check_expected | External | usr1 |
| | | Local_Plan.get_position | | |
| | | Item.create | | |
| | | Place.insert | | |
| | | External.give_place_item | | |
| | warn_warehouse | External.warn_arrival | Request(Redis_R) | req1 |
| | reserve_space | Local_Plan.reserve_space | Network | from_n_ |
| | insert(Place) | | Warehouse | wp |
| | remove(Place) | | Warehouse | wp |
| | get_position(Local_Plan) | | Warehouse | w_lp |
| | reserve_space(Local_Plan) | | Warehouse | w_lp |
| | return_space | | Warehouse | w_lp |

Table 3.1: The OCT for the warehouse system

rules already discussed in Chapter 2. The result in presented in Tables 3.1 and 3.2. The OCD, built from the OCT, is depicted in Figure 3.9.

## Task 3.2: Specify class templates

We now start writing the LOTOS specification. Each node in the OCD is specified by a process and one or more ADTs. We can start by specifying processes first or ADTs first. Then all the processes are put together by using the LOTOS parallel operators and by following the algorithm in [3].

Let us take as an example the class templates which participate in the `Warehouse` aggregation relationship and show how they can be modelled in LOTOS. The aggregate is composed of two hidden components, `Place` and `Local_Plan`, and one shared component, `Item`. The process defining the class template `Local_Plan`, for example, can take the form:

```
process Local_Plan[w_lp](l_plan_id: Local_Plan_id,
                        ids: Item_Place_Id_Set) : noexit :=
    w_lp !get_position !l_plan_id ?ip: Item_Place_Id [ip isin ids];
   Local_Plan[w_lp](l_plan_id, ids)
  []
    w_lp !reserve_space !l_plan_id ?ok: Bool;
```

| Class Templates | Offered Service | Required Service | Clients | Gates |
|---|---|---|---|---|
| Planning | add_redis_req | | Request(Redis_R) | p2 |
| | add_rem_req | | Request(Rem_R) | p2 |
| | get_trans_req | Request.choose_red | Truck | p_tr_ |
| Request | create_i_r(I_R) | | Office | req2 |
| [I_R + Redis_R + Rem_R] | create_red_r(Redis_R) | Planning.add_redis_req | Office, Warehouse | req2 |
| | create_rem_r(Rem_R) | Planning.add_rem_req | Office | req2 |
| | check_expected(I_R) | | Warehouse | req2 |
| | choose_rem(Rem_R) | Warehouse.rm_fetch_items | Planning | p1 |
| | choose_red(Redis_R) | Truck.request_eta | Planning | p1 |
| | | Warehouse.fetch_items | | |
| | | Warehouse.warn_warehouse | | |
| | | Truck.is_delivered | | |
| | | Warehouse.confirm_inserted | | |
| Item | create | | Warehouse | it1 |
| | check_item | | Warehouse, Office | it1 |
| | remove | | Warehouse | it1 |
| | get_place | | Warehouse | it1 |
| | update_item | | Warehouse | it1 |
| | mark_move | | Warehouse, Office | it1 |
| | mark_remove | | Office | it1 |
| Truck | get_trans_req | Planning.get_trans_req | External | usr2 |
| | request_eta | External.get_eta | Request(Redis_R) | tr_r_ |
| | is_delivered | External.confirm_delivered | Request(Redis_R) | tr_r_ |
| Network | reserve_space | Warehouse.reserve_space | Warehouse, Office | to_n_ |

Table 3.2: The OCT for the warehouse system (continued)

```
        Local_Plan[w_lp](l_plan_id, ids)
     []
        w_lp !return_space !l_plan_id;
        Local_Plan[w_lp](l_plan_id, ids)
   endproc (*Local_Plan*)
```

`l_plan_id` and `ids` are attributes of `Local_Plan` and are specified as ADTs. We could have incorporated them into a single ADT, say `Local_Plan_State`, and then use this ADT as a parameter of the process. For simplicity of the exercise we show both attributes individually.

Another example is `Item`:

```
process Item[it1](it: Item_State) : noexit :=
     it1 !check_item !Get_It_Id(it) !Can_Move(it) !Get_Ware(it);
     Item[it1](it)
  []
     it1 !update_item !Get_It_Id(it) ?w: Warehouse_Id ?ip: Item_Place_Id;
     Item[it1](Move(w, ip, it))
  []
     it1 !get_place !Get_It_Id(it) !Get_Place_Id(it);
     Item[it1](it)
  []
     it1 !mark_move !Get_It_Id(it);
     Item[it1](Mark_m(it))
  []
     it1 !mark_remove !Get_It_Id(it);
     Item[it1](Mark_r(it))
  []
     it1 !remove !Get_It_Id(it) !Get_Ware(it) !Get_Place_Id(it);
     stop
endproc (* Item *)
```

The ADT defining `ItemState` is specified as follows:

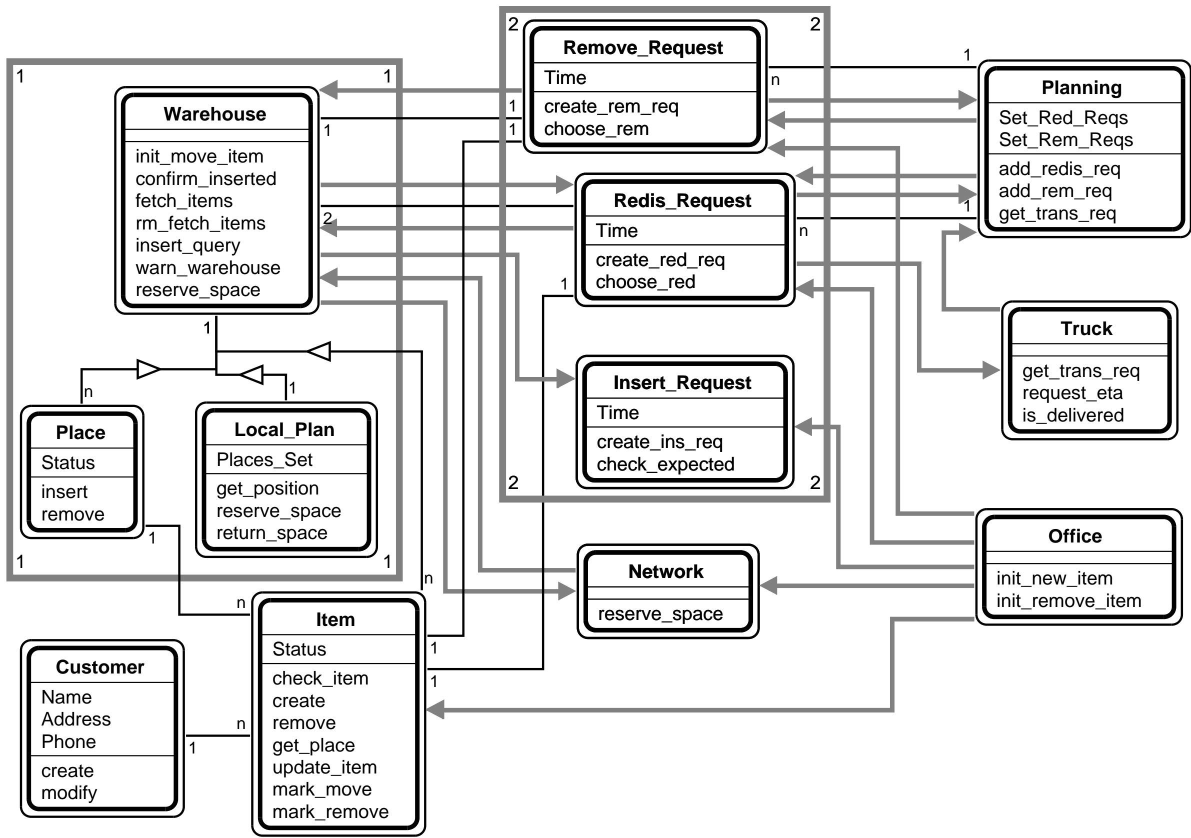



Figure 3.8: Warehouse object model

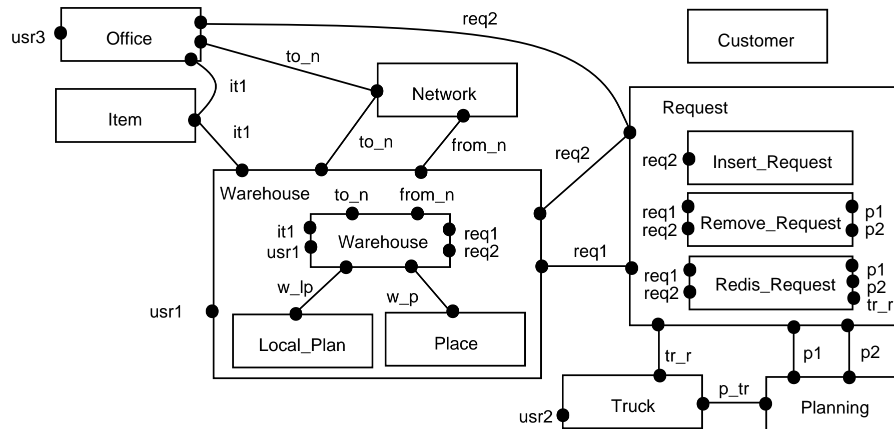



Figure 3.9: Initial OCD for the warehouse system

```
type Item_Type is Item_Id_Set_Type, Item_Place_Id_Set_Type,
                  Warehouse_Id_Type, Boolean
   sorts Item_State
   opns Make_Item : Item_Id, Warehouse_Id, Item_Place_Id, Bool, Bool
                          -> Item_State
        Get_It_Id : Item_State      -> Item_Id
        Get_Ware : Item_State -> Warehouse_Id
```

```
          Get_Place_Id :Item_State -> Item_Place_Id
          Can_Move :Item_State -> Bool
          Move : Warehouse_Id, Item_Place_Id, Item_State -> Item_State
          Mark_m : Item_State -> Item_State
          Mark_r : Item_State -> Item_State
     eqns forall it: Item_Id, ip, nip: Item_Place_Id, w, nw: Warehouse_Id,
                 s1, s2: Bool
       ofsort Item_Id
         Get_It_Id(Make_Item(it, w, ip, s1, s2)) = it;
       ofsort Item_Place_Id
         Get_Place_Id(Make_Item(it, w, ip, s1, s2)) = ip;
       ofsort Warehouse_Id
         Get_Ware(Make_Item(it, w, ip, s1, s2)) = w;
       ofsort Item_State
          Move(nw, nip, Make_Item(it, w, ip, s1, s2))
             = Make_Item(it, nw, nip, true, s2);
          Mark_m(Make_Item(it, w, ip, s1, s2))
             = Make_Item(it, w, ip, false, s2);
          Mark_r(Make_Item(it, w, ip, s1, s2))
             = Make_Item(it, w, ip, s1, false);
       ofsort Bool
          Can_Move(Make_Item(it, w, ip, s1, s2)) = s1 and s2;
   endtype
```

The ADT `Item_Id_Set_Type` is the ADT where the sort of the identifier for objects of the class template `Item` are defined. It is as follows:

```
  type Item_Id_Set_Type is Set_Id_Type
     renamedby
     sortnames Item_Id for Id
               Item_Id_Set for Set
  endtype
```

`Set_Id_Type` is defined as discussed in [11] and is present in the library.

Finally, the aggregate can be defined as:

```
   process Warehouse[usr1, req1, req2, it1, to_n, from_n]
                      (wid: Warehouse_id) : noexit :=
    hide wp, w_lp in
      Warehouse[usr1, req1, req2, it1, wp, w_lp, to_n, from_n](wid)
     |[wp, w_lp]|
      ( Place[wp](..)
      |||
        Local_Plan[w_lp](id14 of Local_Plan_Id, Insert(id7, Insert(id8,
                           Insert(id9, {} of Item_Place_Id_Set))))
      )
  where
    process Warehouse[usr1, req1, req2, it1, wp, w_lp, to_n, from_n]
              (wid: Warehouse_id) : noexit :=
      (  usr1 !insert_query !wid ?ir:Ins_Req_Id ?t:Time;
         req2 !check_expected !ir ?ok:Bool !wid !t;
         usr1 !rtn_insert_query !wid !ok !ir;
         ( [ok] ->
                w_lp !get_position ?lp_id: Local_Plan_Id ?ip:Item_Place_Id;
                it1 !create ?it: Item_id !wid !ip;
                wp !insert !ip !it;
                usr1 !give_place_item !wid !it !ip;
                exit(wid)
           []
```

```
          [not(ok)] ->
               exit(wid)
         )
       []
         usr1 !init_move_item !wid ?it: Item_Id ?t:Time ?dest: Warehouse_Id;
         it1 !check_item !it ?ok:Bool ?w: Warehouse_Id;
         ([not(ok)] -> usr1 !rtn_init_move_item !wid !ok;
                       exit(wid)
         []
          [ok]  -> ([w ne dest] ->
                       to_n !reserve_space ?net:Network_Id !dest;
                       to_n !rtn_reserve_space !net !dest ?is_space:Bool;
                       ([is_space] ->
                            req2 !create_red_r ?id: Red_Req_Id !it !w !dest !t;
                            it1 !mark_move !it;
                            usr1 !rtn_init_move_item !wid !true;
                            exit(wid)
                       []
                        [not(is_space)] ->
                            usr1 !rtn_init_move_item !wid !false;
                            exit(wid)
                       )
                   []
                    [w eq dest] ->
                       usr1 !rtn_init_move_item !wid !false;
                       exit(wid)
                   )
         )
       []
         req1 !warn_warehouse !wid ?t: Time;
         usr1 !warn_arrival !wid !t;
         exit(wid)
       []
         req1 !confirm_inserted !wid ?it:Item_Id;
         w_lp !get_position ?lp_id: Local_Plan_Id ?ip:Item_Place_Id;
         wp !insert !ip !it;
         it1 !update_item !it !wid !ip;
         usr1 !give_place_item !wid !it !ip;
         req1 !rtn_confirm_inserted !wid;
         exit(wid)
       []
         ...
      ) >> accept id: Warehouse_Id
        in Warehouse[usr1, req1, req2, it1, wp, w_lp, to_n, from_n](id)
    endproc (* Warehouse *)
    process Place ... endproc (* Place *)
    process Local_Plan ... endproc (* Local_Plan *)
  endproc (* Warehouse *)
```

As `Place` and `Local Plan` are hidden components, their specification process is internal to the process specifying `Warehouse` (see above). The same does not happen with the aggregate component `Item`. Since it is a shared component, it appears in the LOTOS top-level behaviour expression as an object which is related with the aggregate as a whole. For example:

```
((hide it1, to_n, from_n in
 (...
  |||
   Warehouse[usr1, req1, req2, it1, to_n, from_n](id2 of Warehouse_id)
 )
```

```
|[it1, to_n, from_n]|
 (Item[it1](...) ||| Network[to_n, from_n](id6 of Network_Id))
)
```

### Task 3.3: Compose the objects into a behaviour expression

Having the class templates specified we can compose them to form the top-level behaviour expression.

```
 (hide req1, req2, usr1, usr2, usr3 in
    ((hide it1, to_n, from_n in
     ( Office[usr3, req2, it1, to_n, from_n](id1 of Office_Id)
       |||
       Warehouse[usr1, req1, req2, it1, to_n, from_n](id1 of Warehouse_id)
       |||
       Warehouse[usr1, req1, req2, it1, to_n, from_n](id2 of Warehouse_id)
     )
     |[it1, to_n, from_n]|
     (Item[it1](...) ||| Network[to_n, from_n](id6 of Network_Id))
    )
    |[req1, req2]|
    (hide p1, p2, tr_r in
      Request[req1, req2, p1, p2, tr_r]
      |[p1, p2, tr_r]|
      (hide p_tr in
        Truck[usr2, tr_r, p_tr](id1 of Truck_Id)
        |[p_tr]|
        Planning[p1, p2, p_tr](id1 of Plan_Id, {} of Red_Req_Id_Set,
                                {} of Rem_Req_Id_Set)
      )
    )
  )
|[usr1, usr2, usr3]|
 Interface_Scenario[usr1, usr2, usr3])
```

### Task 3.4: Prototype the specification

To prototype the specification, we use the LITE tools. Also, we describe in LOTOS the interface scenarios so that we can required services from the system and receive the respective answers.

An interface scenario to check part of the functionality of the system could be defined as follows:

```
process Interface_Scenario[usr1, usr2, usr3] : noexit :=
 (* Initialize the system *)
  usr3 !init_new_item ?oid:Office_Id !This_Time ?w: Warehouse_Id;
  usr3 !rtn_init_new_item !oid ?ins_req_id: Ins_Req_Id !true;
  usr1 !insert_query !w !ins_req_id !This_Time;
  usr1 !rtn_insert_query !w !true !ins_req_id;
  usr1 !give_place_item !w ?it_id: Item_Id ?ip: Item_Place_Id;
  usr1 !init_move_item !w !it_id !This_Time ?w2: Warehouse_Id [w ne w2];
  usr1 !rtn_init_move_item !w !true;
  usr2 !get_trans_req ?tr_id: Truck_Id;
  usr2 !get_eta !tr_id !This_Time ?wh: Warehouse_Id ?it1_id: Item_Id;
  usr1 !fetch !wh ?it2_id: Item_Id ?pl_id: Item_Place_Id;
  usr2 !get_eta !tr_id !This_Time ?wh2: Warehouse_Id !it1_id;
  usr1 !warn_arrival !w2 ?t: Time;
  usr2 !confirm_delivered !tr_id !it1_id !wh2;
  usr1 !give_place_item !w2 ?it2_id: Item_Id ?ip2: Item_Place_Id;
```

```
    usr3 !init_remove_item !oid !it_id !This_Time !w2;
    usr3 !rtn_init_remove_item !oid !true;
    usr1 !fetch !w2 ?it3_id: Item_Id ?pl2_id: Item_Place_Id;
    (hide success in success; stop)
  endproc (* Interface_Scenario *)
```

## Task 3.5: Refine the specification

To refine our specification we model static relationships (we have already modelled some of them), we introduce object generators, so that we can deal with multiple instances of class templates, we identify higher level objects, we take final decisions about whether or not a given object should be modelled as a process or as an ADT, and we refine each process and ADT by adding more detail, if necessary.

The main decision here is to define object generators for some of the class templates. We decided that the class templates `Office`, `Network`, `Planning` and `Local_Plan` do not need an object generator. The final OCD is depicted in Figure 3.10

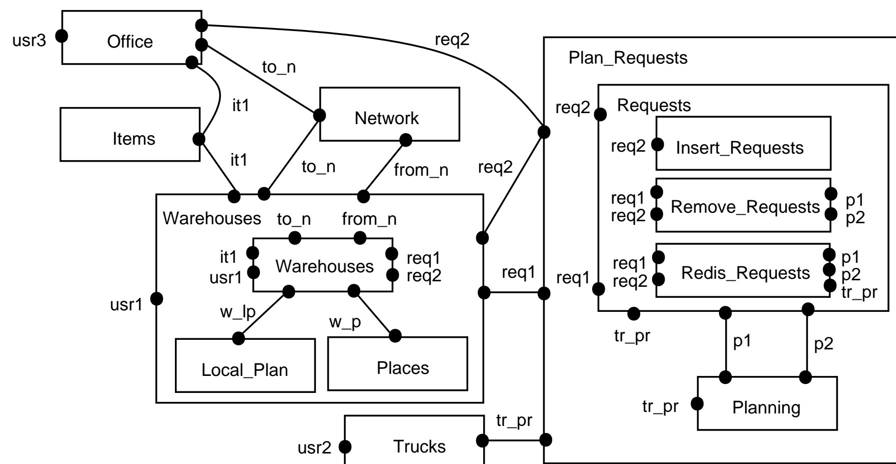



Figure 3.10: Final OCD

An example of an object generator is the process `Items`:

```
process Items[it1](ids: Item_Id_Set) : noexit :=
   it1 !create ?id: Item_Id ?w: Warehouse_Id ?ip: Item_Place_Id
          [(id notin ids)];
   ( Item[it1](Make_Item(id, w, ip, true, true))
    |||
     Items[it1](Insert(id, ids))
   )
  []
    it1 !check_item ?id: Item_Id !false ?w: Warehouse_Id [(id notin ids)];
    Items[it1](ids)

 where
 process Item[it1](it: Item_State) ... endproc (* Item *)
 endproc (* Items *)
```

Sending a `create` message to `Items` causes a new `Item` to be created. `Items` holds the set of item identifiers already allocated and the selection predicate `[(id notin ids)]` guarantees that each `id` generated is different from the existing ones. The event

```
      it1 !check_item ?id: Item_Id !false ?w: Warehouse_Id [(id notin ids)];
```

catches the situations where an attempt is made to synchronize with a non-existent item.

## 3.3 Conclusions

This chapter discussed the application of ROOA to the warehouse management problem present by Jacobson [9]. We found many difficulties in understanding Jacobson's object model. In our opinion a class template cannot be fully understood until we define its services and attributes (although Jacobson does not include them). We used the use cases to identify the information that was missing in the object model, but this was a difficult task.

Jacobson includes in his object model a system of window interfaces and objects which were part of the environment rather than the system. We eliminated some of those objects, for example `Warehouse Truck Radio` and `Truck Radio`, as they were not of much use for our exercise.

The warehouse management system has a more complex pattern of communication than the banking system discussed in the previous chapter. This led us to make more use of event trace diagrams, as an auxiliary technique to understand and refine the object model.

Also, this problem can be seen as complementary to the banking system, since it deals with the other interesting object-oriented concept for structuring systems. The banking system focused on inheritance while the warehouse focused on aggregation. Aggregation with either hidden or shared components is easily modelled in LOTOS, as can be seen from this chapter. (For more information on aggregation, please see [11, 15].)

The LOTOS specification for the warehouse management system is given in Appendix B.

# Chapter 4

# A car rental system

## 4.1 Introduction

This chapter illustrates one more case study, using ROOA. The problem is to analyse a simple car rental system. The car rental system was first proposed as a final example to check the general characteristics of ROOA. Therefore, in this document we only discuss the more interesting functionality of the problem.

## 4.2 The car rental system requirements

An outline of the requirements is:

> A car rental company wishes to introduce a new automatic rental and billing system. The hardware has already been chosen and consists of a central computer which operators access via terminals at the different branches of the company.
>
> Each terminal is positioned in a branch and a branch may have more than one terminal.
>
> Vehicles may be booked in advance or rented immediately. Booking and renting occur via the terminals. A vehicle may only be rented at a terminal positioned in the branch from which the vehicle is to be taken, but it may be booked at any terminal. Vehicles may be taken from one branch and returned to another, provided that this is agreed when the rental contract is issued.
>
> Clients can be corporate clients or ordinary clients. Corporate clients (companies) have accounts with the car rental firm. Bills for such rentals are sent to the company after the car is returned and do not have to be paid for directly by the person renting the car. For ordinary clients:
>
> - When a vehicle is booked in advance, a deposit must be paid. If the advance booking is cancelled, the deposit is forfeited.
> - When a vehicle is rented the normal costs have to be paid.
> - When a vehicle is returned, any extra costs (e.g. mileage excess) must be paid.
>
> If an ordinary client has not rented a car before, they must prove their identity and show a driving licence before driving away the car. This information is then stored. This is not necessary when it is a known client although information about the new rental will be added to the information held. If a previous client is a bad risk then such information will be held and their custom will be refused.
>
> The system keeps a list of the types of car on offer – model, engine capacity, number of doors, automatic or manual transmission, daily cost of rental. Car types are grouped

into price classes. The system must keep track of the actual cars available at the different branches.

When a client wishes to book or rent a car, they are given a list of up to two cars which meet their requirements and which are available. They then decide on one of the cars or decline the offer during that terminal transaction. If a car is accepted, a rental contract with the client is generated. Multiple terminals are available. Each terminal will only deal with one customer at a time. While a car is being offered to one client at a terminal, it cannot be offered to another client at another terminal.

When a car is returned, this is noted. The car is not available for rental until after it has been checked for defects.

The system must allow terminals, vehicle types, actual cars and company accounts to be added or removed from the system. It must also be possible to record the movement of a car from one branch to another.

The system must keep a record of all transactions so that statistics can be produced on, for example, the performance of each branch.

## 4.3 Applying ROOA

### Task 1: Build an object model

The first task is to apply any of the existing OOA methods to create an object model. An initial object model created by following Coad and Yourdon's method is depicted in Figure 4.1.

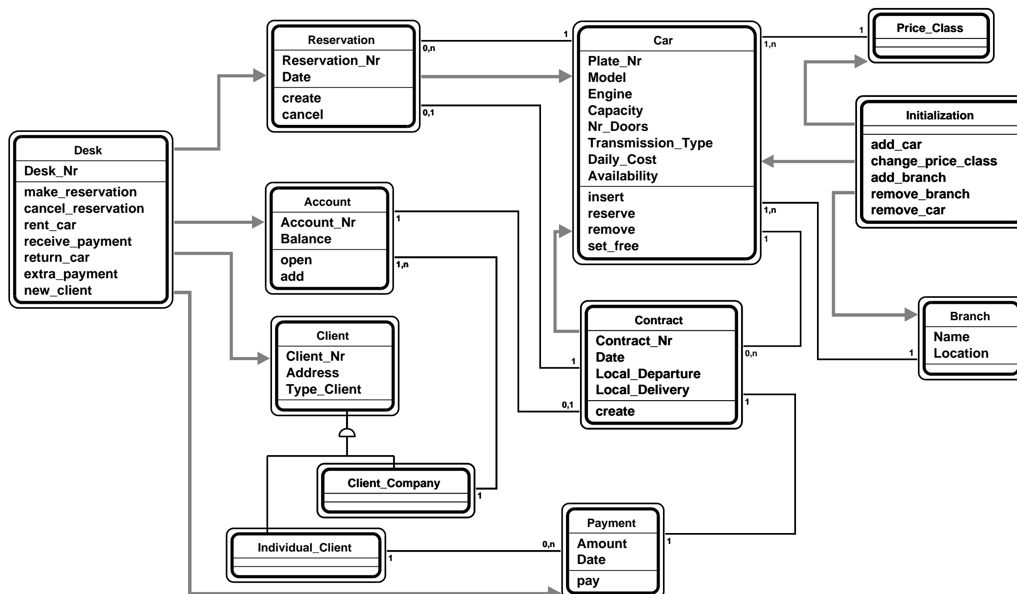



Figure 4.1: Initial object model

### Task 2: Refine the object model

This task is composed of three subtasks: complete the object model, initial identification of dynamic behaviour, structure the object model.

**Task 2.1: Complete the object model**

This subtask guarantees that the object model contains interface objects, static relationships and attributes. The object model presented in Figure 4.1 already contains this information.

**Task 2.2: Initial identification of dynamic behaviour**

This subtask plays special attention to the dynamic behaviour of each class template and of the system as a whole.

Here we define interface scenarios. Interface scenarios help us identify the services required from the system. Also, they help us navigate through the object model in order to identify services and message connections. Each interface scenario will later be translated to LOTOS; now we represent it textually. During this study we identified three main transactions: make a reservation, rent a car, return the car. We define one interface scenario for each transaction. Table 4.1 shows an example of an interface scenario.

| A client goes to the desk and asks for a car reservation;<br>(The client gives information about the kind of car desired.)<br>If the reservation succeeded, the client pays the deposit and receives a receipt.<br>If it is a new client, a new client record is created. |
|---|

Table 4.1: Interface scenario for Reservation

This interface scenario only shows the interface between the clients and the system. We use event trace diagrams to show what happens inside the system when the interface scenario requires a service. Figure 4.2 shows the event trace diagram which corresponds to making a reservation (and cancelling it). Figure 4.3 shows the event trace diagram which corresponds to renting a car,

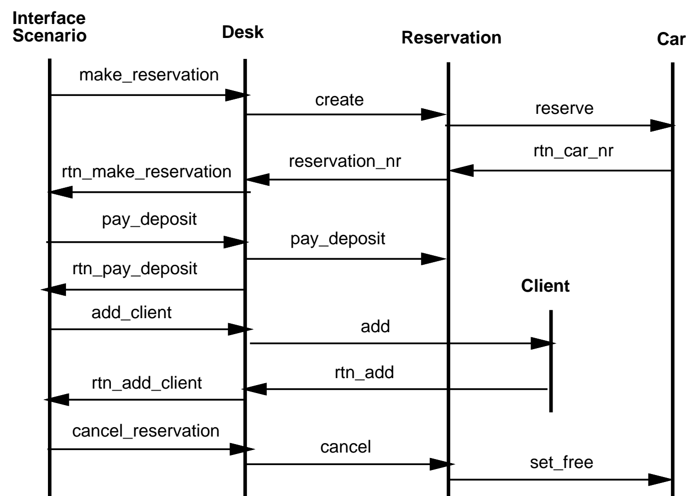



Figure 4.2: Event trace diagram for reservation transaction

with or without a reservation. Figure 4.4 shows the event trace diagram which corresponds to the delivery of the car.

As we build the event trace diagrams and interface scenarios, we can collect the information in an OCT. Some of the information was already in the initial object model. However, the object model is changed as we understand the system better from the information gained by defining interface scenarios and event trace diagrams. The initial OCT is shown in Table 4.2.

The creation of the interface scenarios, event trace diagrams and OCTs should be done in parallel. Each one behaves as a check list for the others, and they help validate each other.

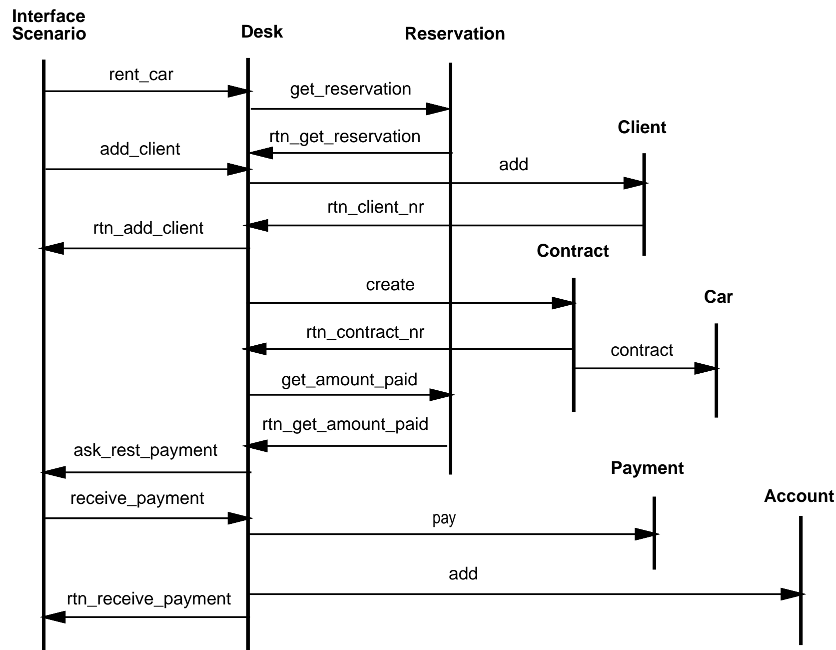



Figure 4.3: Event trace diagram for rental transaction

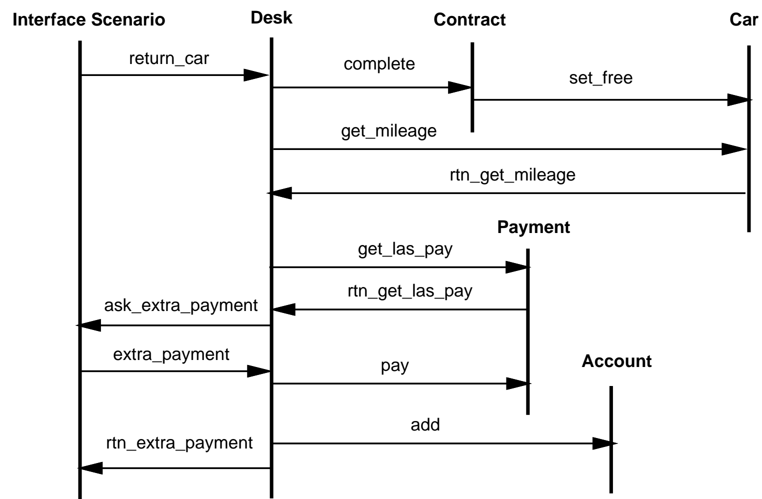



Figure 4.4: Event trace diagram for return transaction

### Task 2.3: Structure the object model

In this subtask we propose subsystems and we look for complex relationships such as inheritance and aggregation. We have already identified one inheritance relationship, namely `Client`. As ROOA proposes, this is a good candidate for a subsystem. We could then group `Client`, `Client_Company` and `Individual_Client` to form the subsystem `Customer`.

After the first iteration, and based on the information contained in the event trace diagrams, and OCT, the object model is refined to take the form illustrated in Figure 4.5.

| Class Templates | Offered Service | Required Service | Clients | Gates |
|---|---|---|---|---|
| Desk | make_reservation | Reservation.create | External | d |
| | pay_deposit | Reservation.pay_deposit | External | d |
| | cancel_reservation | Reservation.cancel | External | d |
| | rent_car | Reservation.get_reservation | External | d |
| | | Client.add | | |
| | | Contract.create | | |
| | | Reservation.get_amount_paid | | |
| | receive_payment | Account.add | External | d |
| | | Payment.pay | | |
| | return_car | Contract.complete | External | d |
| | | Car.get_mileage | | |
| | | Payment.get_last_payment | | |
| | | Account.get_last_payment | | |
| | extra_payment | Payment.pay | External | d |
| | | Account.add | | |
| | add_client | Client.add | External | d |
| Reservation | create | Car.reserve | Desk | r |
| | cancel | Car.set_free | Desk | r |
| | pay_deposit | | Desk | r |
| | get_amount_paid | | Desk | r |
| | get_reservation | | Desk | r |
| Contract | create | | Desk | c |
| | complete | Car.set_free | Desk | c |
| Car | insert | | Initialization | c1 |
| | remove | | Initialization | c1 |
| | reserve | | Reservation | c2 |
| | contract | | Contract | c2 |
| | set_free | | Reservation, Contract | c2 |
| | get_mileage | | Desk | ? |
| Branch | insert | | Initialization | b |
| | remove | | Initialization | b |
| Initialization | add_car | Car.insert | External | n |
| | remove_car | Car.remove | External | n |
| | add_branch | Branch.insert | External | n |
| | remove_branch | Branch.remove | External | n |
| | change_price_class | Price_Class.change | External | n |
| | add_price_class | Price_Class.add | External | n |
| | remove_price_class | Price_Class.remove | External | n |
| Account | open | | Desk | a |
| | add | | Desk | a |
| | get_last_payment | | Desk | a |
| Payment | pay | | Desk | p |
| | get_last_payment | | Desk | p |
| Price_Class | add | | Initialization | pc |
| | change | | Initialization | pc |
| | remove | | Initialization | pc |
| Client | remove | | Desk | cl |
| Client_Company | add | | Desk | cl |
| Individual_Client | add | | Desk | cl |

Table 4.2: An OCT for the car rental system

## Task 3: Build the LOTOS formal model

In the first iteration, we need to accomplish four main subtasks: create an object communication diagram, specify class templates, compose the objects into a behaviour expression and prototype the specification. The following subsections show the results.

## Task 3.1: Create an Object Communication Diagram (OCD)

From the OCT we can draw an OCD which shows the structure of the object model and, at the same time, shows the communications between the objects in the system.

In the first iteration each node in the OCD represents an object. Figure 4.6 depicts an OCD for the problem we are studying.

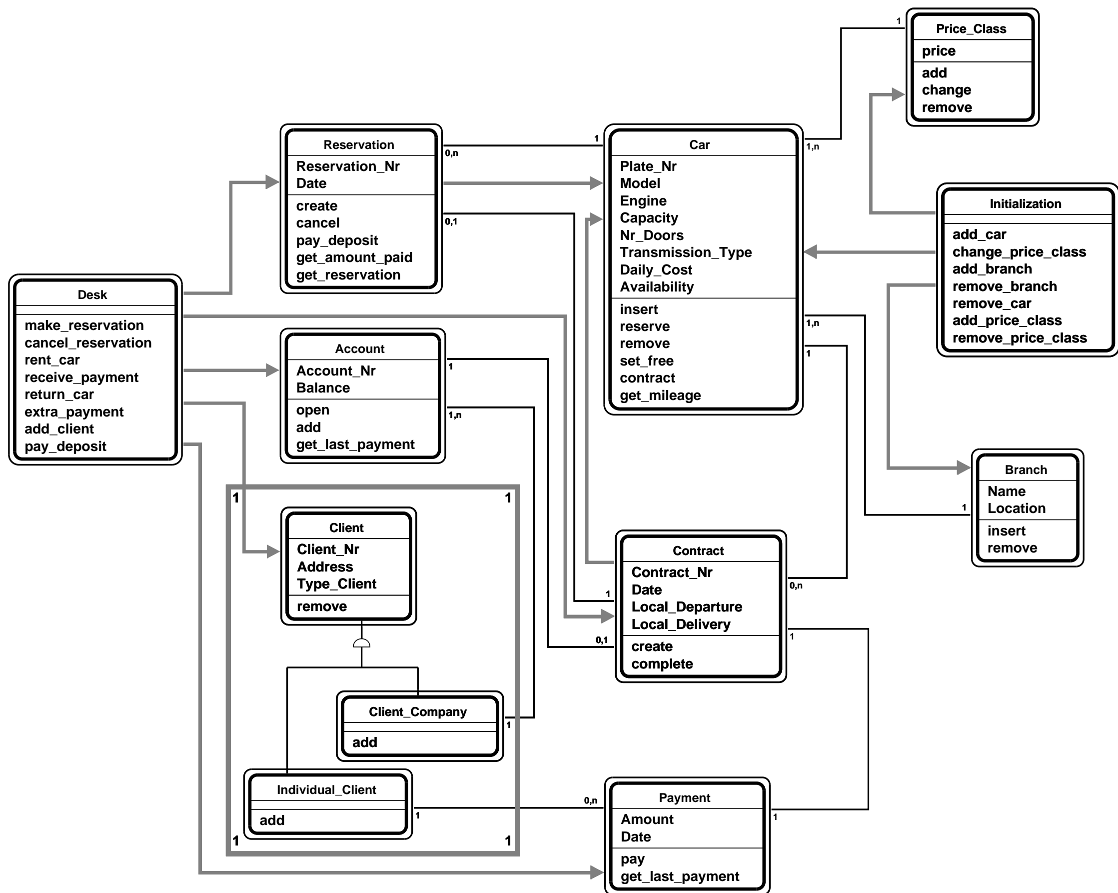



Figure 4.5: Final object model

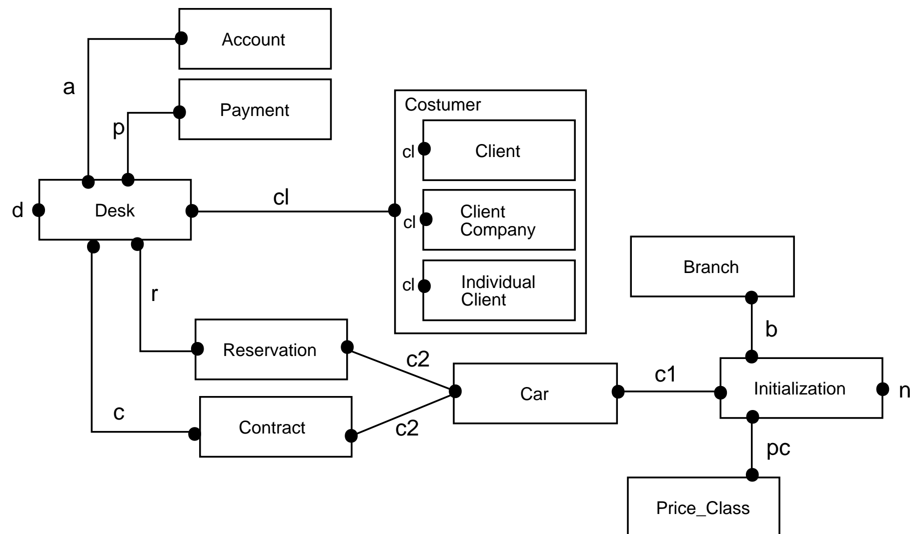



Figure 4.6: An OCD for the car rental system

## Task 3.2: Specify class templates

In general, a class template is specified by using a process and one or more ADTs. An example of a class template is `Car`:

```
process Car[c2](this_car: State_Car): noexit :=
  ( c2 !reserve !Get_Car_Id(this_car);
    exit(Change_State(this_car, reserved of Availability))
  []
    c2 !set_free !Get_Car_Id(this_car);
    exit(Change_State(this_car, free of Availability))
  []
    c2 !contract !Get_Car_Id(this_car);
    exit(Change_State(this_car, contracted of Availability))
  []
    c2 !get_mileage !Get_Car_Id(this_car) !Get_Mileage(this_car);
    exit(this_car)
  ) >> accept this_car: State_Car in Car[c2](this_car)
 endproc (* Car *)
```

And the ADT where the sort `State Car` defined, is specified as follows:

```
type Car_Type is Car_Id_Set_Type, Availability_Type, Mileage_Type
  sorts State_Car
  opns
    Make_Car     : Car_Id,  Availability, Mileage -> State_Car
    Change_State : State_Car, Availability        -> State_Car
    Get_Mileage  : State_Car -> Mileage
    Get_Car_Id   : State_Car -> Car_Id
eqns forall c: State_Car, n: Car_Id, v: Availability, m: Mileage
  ofsort Car_Id
    Get_Car_Id(Make_Car(n,v,m)) = n;
    Get_Car_Id(Change_State(c,v)) = Get_Car_Id(c);
  ofsort Mileage
    Get_Mileage(c) = Some_Mileage;
endtype
```

### Task 3.3: Compose the objects into a behaviour expression

From the OCD depicted in Figure 4.6, and by following the algorithm presented in [3], we can build a behaviour expression which shows how all the objects in the system interact with each other.

```
 hide d, c, c1, c2, r in
  (( Desk [d, r, c](id1 of Desk_Id)
      |[r, c]|
       ( Reservations[r, c2]({} of Reservation_Id_Set)
       |||
         Contracts[c, c2]({} of Contract_Id_Set)
       )
    )
   |[c1, c2]|
     Cars[c1, c2]({} of Car_Id_Set)
  )
|[d]|
  Interface_Scenario[d]
(...)
```

### Task 3.4: Prototype the specification

As with the specifications discussed in Chapters 2 and 3, we use the Lite tools to prototype the specification created for the car rental system. This helps us find syntax and semantic errors and enables us to execute the specification to validate the results against the requirements.

### Task 3.5: Refine the specification

As we said before, we used this exercise to test the general characteristics of ROOA. Therefore, we have not been interested in refining and completing the final specification.

Among the list of tasks we can accomplish here we specified conceptual relationships between objects and introduced object generators.

An example of an object generator for class template `Car`, is as follows:

```
process Cars[c1, c2](cs: Car_Id_Set): noexit :=
  ( c1 !create ?c_id: Car_Id [c_id notin cs];
    ( Car[c2](Make_Car(c_id, free of Availability, Some_Mileage))
    |||
      Cars[c1, c2](Insert(c_id, cs))
    )
  )
  where

 process Car ... endproc
```

## 4.4 Conclusions

The car rental system was proposed to check the general characteristics of ROOA, and not to analyse the details of a specification (for that purpose we already have the banking system and the warehouse management problem). This is why, in this document, we only discuss the functionality of the problem which involved communication between a large number of objects.

Applying ROOA to this problem was an useful exercise, to make us feel more comfortable with our method. We tested how the various ROOA techniques interact and help each other as validation tools. In particular, we tested each task and subtask. The car rental system has some interesting communication patterns. This helped us to evaluate once more the need for the event trace diagrams, as an intermediate technique to help build the object model. Once more, the event trace diagrams were also very useful in helping us build the object communication tables which then were the major source of information in the creation of the object communication diagram.

After applying ROOA to this problem we have drawn an important conclusion: object model, event trace diagrams, object communication tables, object communication diagram and the LOTOS specification were being used many times to improve each other. All these techniques are integrated in the sense that, at a given stage, we were using all of them at the same time, in a parallel and interactive way. For example, from the event trace diagrams and the object-communication table we started writing the LOTOS specification. As this advanced, we identified more services and communications which had not been fully identified in the previous tasks.

A simple version of the LOTOS specification for this problem is presented in Appendix C.

# Chapter 5

# Conclusions

ROOA enables a formal object-oriented analysis model to be devised from a set of informal requirements, and results in a formal requirements specification expressed in LOTOS. ROOA consists of three main tasks: building an object model, refining the object model, and building the formal LOTOS OOA model. Each of these tasks involves multiple passes through subtasks. The three tasks are not necessarily sequential: some parts of the model may be built through to the LOTOS specification before other parts of the model are analysed.

The first task, building the object model, may be accomplished in the first pass by using any of the usual object-oriented analysis methods, such as the methods of Coad and Yourdon [5], Rumbaugh *et al.* [17] and Jacobson [9]. The object model is refined in the second task by passing through three subtasks: completing the object model, identifying the initial identification of dynamic behaviour, structuring the object model. The formal LOTOS OOA model integrates the static, dynamic and functional properties of the system, and consists of five subtasks: creating the object communication diagram (OCD); specifying the class templates as LOTOS processes and ADTs; composing objects; prototyping the object model by executing the LOTOS specification; and refining the specification according to the results of this rapid prototyping. (For a full description of the ROOA method, please see [11, 12, 13, 14, 16].)

This document showed how ROOA can be applied to a problem so that an initial formal requirements specification can be created. ROOA has been applied to database oriented problems with simple communication patterns and to problems with a more complex dynamic behaviour. Also, it was used together with the methods by Rumbaugh *et al.* [17], Jacobson [9] and Coad and Yourdon [5]. This helped us identify system development requirements and object models with which ROOA had be compatible.

We have discussed three problems: an automated banking system, a warehouse management system and a car rental system. The banking system and the warehouse were the two most interesting problems, so we have discussed them in more detail.

The banking system is basically a database-oriented problem, with a simple dynamic behaviour. It was important to show how ROOA models class templates, creates multiple instances of a class template and deals with inheritance.

The warehouse management system was first described and analysed by Jacobson [9]. This problem has a more complex dynamic behaviour. It shows the advantage of using event trace diagrams within ROOA. Another important object-oriented concept dealt with in this problem is aggregation.

The car rental system is mainly a dynamic-oriented problem, but it is not as interesting as the warehouse system.

Each of these problems deal with issues which are important when specifying systems, mainly, the static versus dynamic problems and inheritance versus aggregation. In this document we showed how ROOA behaves when applied to these kind of problems. Because ROOA uses LOTOS, it is specially good at describing the dynamic behaviour of systems, but also, it shows the static structure of a system.

The LOTOS specifications for the three problems, the banking system, the warehouse system and the car rental system, are given in Appendices A, B and C.

# Bibliography


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

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

[3] 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.

[4] R.G. Clark and A.M.D. Moreira. Behavioural Inheritance in ROOA. In R. Wieringa and R. Feenstra, editors, *Workshop on Information Systems - Correctness and reusability (IS-CORE'94)*, pages 346–356, Amsterdam, The Netherlands, September 1994.

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

[6] 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.

[7] C. Gane and T. Sarson. *Structured Systems Analysis: Tools and Techniques.* Prentice-Hall, 1979.

[8] ISO. Information Processing Systems – Open Systems Interconnection – LOTOS : A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, International Standard 8807. ISO, 1988.

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

[10] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[11] A.M.D. Moreira. *Rigorous Object-Oriented Analysis.* PhD thesis, Department of Computing Science and Mathematics, University of Stirling, Scotland, August 1994.

[12] A.M.D. Moreira and R.G. Clark. LOTOS in the Object-Oriented Analysis Process. In *BCS-FACS Workshop on Formal Aspects of Object-Oriented Systems*, Imperial College, London, December 1993. *BCS-FACS (British Computer Society – Formal Aspects of Computing Science).*

[13] A.M.D. Moreira and R.G. Clark. ROOA: Rigorous Object-Oriented Analysis. Technical Report CSM-109, Department of Computing Science and Mathematics, University of Stirling, Scotland, October 1993.

[14] A.M.D. Moreira and R.G. Clark. Combining Object-Oriented Analysis and Formal Description Techniques. In M. Tokoro and R. Pareschi, editors, *ECOOP'94*, volume 821 of *Lecture Notes in Computer Science*, pages 344–364. Springer-Verlag, 1994.

[15] A.M.D. Moreira and R.G. Clark. Complex Objects: Aggregates. Technical Report CSM-123, Department of Computing Science and Mathematics, University of Stirling, Scotland, May 1994.

[16] A.M.D. Moreira and R.G. Clark. Rigorous Object-Oriented Analysis. In E. Bertino and S. Urban, editors, *International Symposium on Object-Oriented Methodologies and Systems (ISOOMS)*, volume 858 of *Lecture Notes in Computer Science*, pages 65–78. Springer-Verlag, 1994.

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

[18] S. Shlaer and S.J. Mellor. An Object-Oriented Approach to Domain Analysis. *ACM Software Engineering Notes*, 14(5):66–77, July 1989.

# Appendix A

# LOTOS specification for the banking system

```
specification Automated_Bank [t, ob1]: noexit

  library
    Boolean, NaturalNumber, Set
  endlib

  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
         h   : Id -> Nat
         _eq_, _ne_, _lt_ : Id, Id -> Bool
         First_Set  : Id -> Bool
         Second_Set : Id -> Bool
         Third_Set  : Id -> Bool
         Fourth_Set : Id -> Bool
    eqns forall n1, n2: Id
      ofsort Nat
        h(id1) = 0;
        h(id2) = succ(h(id1));
        h(id3) = succ(h(id2));
        h(id4) = succ(h(id3));
        h(id5) = succ(h(id4));
        h(id6) = succ(h(id5));
        h(id7) = succ(h(id6));
        h(id8) = succ(h(id7));
        h(id9) = succ(h(id8));
        h(id10) = succ(h(id9));
        h(id12) = succ(h(id11));
        h(id13) = succ(h(id12));
        h(id14) = succ(h(id13));
        h(id15) = succ(h(id14));
        h(id16) = succ(h(id15));
        h(id17) = succ(h(id16));
        h(id18) = succ(h(id17));
        h(id19) = succ(h(id18));
        h(id20) = succ(h(id19));
```

```
      ofsort Bool
        n1 eq n2 = h(n1) eq h(n2);
        n1 ne n2 = h(n1) ne h(n2);
        n1 lt n2 = h(n1) lt h(n2);
        First_Set(n1)  = h(n1) lt h(id5);
        Second_Set(n1) = not(h(n1) lt h(id5))  and (h(n1) lt h(id10));
        Third_Set(n1)  = not(h(n1) lt h(id10)) and (h(n1) lt h(id15));
        Fourth_Set(n1) = not(h(n1) lt h(id15)) and (h(n1) lt h(id20));
  endtype

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

### Abstract data types

Although there is a library where data types are predefined, almost all data types in LOTOS must be specified. All the following data types are defined based on the ones exported by the library. Each object needs an identifier to allow it to be referenced by other objects or by the external world. Instead of defining from scratch an identifier for each object, we have included the abstract data types `Id Type` and `Set Id Type` to be used as a starting point. The `Id Type` is an identifier definition. The `Set Id Type` will be used to define sets of `Id Type`. The definition of the new abstract data types is made by renaming the type defined in the library.

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

  type Client_Number_Set_Type is Set_Id_Type
     renamedby
     sortnames Client_Number     for Id
               Client_Number_Set for Set
  endtype

  type Card_Number_Set_Type is Set_Id_Type
     renamedby
     sortnames Card_Number     for Id
               Card_Number_Set for Set
  endtype

  type Cheque_Number_Set_Type is Set_Id_Type
     renamedby
     sortnames Cheque_Number     for Id
               Cheque_Number_Set for Set
  endtype

  type SO_Number_Set_Type is Set_Id_Type
     renamedby
     sortnames SO_Number for Id
               SO_Number_Set for Set
  endtype
```

```
  type Bank_Name_Set_Type is Set_Id_Type
     renamedby
     sortnames Bank_Name     for Id
               Bank_Name_Set for Set
     opnnames  This_Bank      for id1
  endtype

  type Id_Tellers_Set_Type is Set_Id_Type
     renamedby
     sortnames Id_Tellers     for Id
               Id_Tellers_Set for Set
     opnnames Is_ATM          for First_Set
              Is_CT            for Second_Set
  endtype
(*---------------------------------------------------------------------
```

The Account Number Set Type, Client Number Set Type, Cheque Number Set Type, Card Number Set Type, SO Number Set Type, Bank Name Set Type and Id Tellers Set Type are renamed types of the type Set Id Type. Is Cheque Acc and Is Savings Acc are boolean operations that are going to be used to decide if a given account number is a chequing account or a savings account. Is ATM and Is CT are equivalent operations for the tellers.

```
---------------------------------------------------------------------*)
  type Kind_Account_Type is
     sorts Kind_Account
     opns saving, chequing : -> Kind_Account
  endtype

  type Money_Type is
     sorts Money
     opns This_Mon : -> Money
  endtype

  type Interest_Type is
     sorts Interest
     opns This_Inter : -> Interest
  endtype

  type Period_Type is
     sorts Period
     opns This_Prd : -> Period
          Update_Date: Period -> Period
  endtype

  type Date_Type is
     sorts Date
     opns This_Date : -> Date
  endtype

  type Balance_Type is
     sorts Balance
     opns Some_Balance : -> Balance
  endtype
(*---------------------------------------------------------------------
```

In each of the above types we have defined constants. In the Bank Type we are going to define the equation

```
Get_Bank_Name(Init_Bank(bk)) = bk
```

which give us the name of a bank. The value `This Bank` is going to be used as a constant that gives our bank name.

```
----------------------------------------------------------------------*)
  type Bank_Type is Bank_Name_Set_Type
     sorts Bank
     opns Init_Bank     : Bank_Name -> Bank
          Get_Bank_Name : Bank      -> Bank_Name
     eqns forall bk: Bank_Name
       ofsort Bank_Name
         Get_Bank_Name(Init_Bank(bk)) = bk;
  endtype
(*----------------------------------------------------------------------
```

For a standing order we need to define some more equations. As our aim in this phase is only to introduce the services together with their functionality, we will not show full calculations, since we believe that that is too much detail for the analysis phase.

Notice that the relationships between `Standing Order` and `Account` and between `Standing Order` and `Other Banks` (Figure2.17) are modelled in the abstract data type. The constructor

```
Make_SO: SO_Number, Account_Number,
         Account_Number, Bank_Name, Money -> State_Standing_Order
```

shows that.

```
----------------------------------------------------------------------*)
  type Standing_Order_Type is SO_Number_Set_Type, Money_Type,
                              Account_Number_Set_Type, Bank_Name_Set_Type
     sorts State_Standing_Order
     opns Make_SO: SO_Number, Account_Number, Account_Number,
                   Bank_Name, Money           -> State_Standing_Order
          Get_SO_Number       : State_Standing_Order -> SO_Number
          Get_Paying_Client   : State_Standing_Order -> Account_Number
          Get_Receiving_Client: State_Standing_Order -> Account_Number
          Get_Bank_Name       : State_Standing_Order -> Bank_Name
          Get_Payable_Amount  : State_Standing_Order -> Money
     eqns forall s: SO_Number, n1, n2: Account_Number, b: Bank_Name, a: Money
       ofsort SO_Number
         Get_SO_Number(Make_SO(s, n1, n2, b, a)) = s;
       ofsort Account_Number
         Get_Paying_Client(Make_SO(s, n1, n2, b, a))    = n1;
         Get_Receiving_Client(Make_SO(s, n1, n2, b, a)) = n2;
       ofsort Bank_Name
         Get_Bank_Name(Make_SO(s, n1, n2, b, a)) = b;
       ofsort Money
         Get_Payable_Amount(Make_SO(s, n1, n2, b, a)) = a;
  endtype
(*----------------------------------------------------------------------
```

The relationship between `Cheque` and `Account` is modelled in the same way as the one between `Standing Order` and `Account`.

```
----------------------------------------------------------------------*)
  type Cheque_Type is Cheque_Number_Set_Type, Account_Number_Set_Type,
                      Bank_Name_Set_Type, Money_Type
     sorts State_Cheque
     opns Make_Cheque : Cheque_Number, Account_Number, Account_Number,
```

```
                     Bank_Name, Money         -> State_Cheque
          Get_Cheque_Number: State_Cheque        -> Cheque_Number
          Get_Acc_To       : State_Cheque        -> Account_Number
          Get_Acc_From     : State_Cheque        -> Account_Number
          Get_Amount       : State_Cheque        -> Money
          Get_Bank         : State_Cheque        -> Bank_Name
     eqns forall ch: Cheque_Number, n1, n2: Account_Number, B: Bank_Name, m: Money
       ofsort Cheque_Number
         Get_Cheque_Number(Make_Cheque(ch, n1, n2, b, m)) = ch;
       ofsort Account_Number
         Get_Acc_To(Make_Cheque(ch, n1, n2, b, m)) = n2;
         Get_Acc_From(Make_Cheque(ch, n1, n2, b, m)) = n1;
       ofsort Money
         Get_Amount(Make_Cheque(ch, n1, n2, b, m)) = m;
       ofsort Bank_Name
         Get_Bank(Make_Cheque(ch, n1, n2, b, m)) = b;
  endtype
(*------------------------------------------------------------------------
```

As it would be expected, the Account abstract data type is the most complex, since everything we do in our system affect accounts. The dummy equations we are going to define correspond to the services showed in the object model in Figure 2.17, plus the constructor and some selectors.

```
------------------------------------------------------------------------*)
 type Account_Type is Account_Number_Set_Type, Money_Type,
                      Balance_Type, Date_Type, Interest_Type,
                      Client_Number_Set_Type, Cheque_Number_Set_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
       Credit_Pending     : State_Account, Money     -> State_Account
       Add_Credit_Pending : State_Account, Money     -> State_Account
       Sub_Credit_Pending : State_Account, Money     -> State_Account
       Credit_Interest    : 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, b: Balance
       ofsort Account_Number
         Get_Account_Number(Make_Account(n, b)) = n;
         Get_Account_Number(Credit_Account(a, m)) = Get_Account_Number(a);
         Get_Account_Number(Debit_Account(a, m)) = Get_Account_Number(a);
         Get_Account_Number(Credit_Pending(a, m)) = Get_Account_Number(a);
         Get_Account_Number(Add_Credit_Pending(a, m)) = Get_Account_Number(a);
         Get_Account_Number(Sub_Credit_Pending(a, m)) = Get_Account_Number(a);
         Get_Account_Number(Credit_Interest(a, m)) = Get_Account_Number(a);
       ofsort Balance
         Get_Balance(a) = Some_Balance;
  endtype

  type Client_Type is Client_Number_Set_Type, Card_Number_Set_Type,
                      Account_Number_Set_Type
     sorts Client
     opns Make_Client : Client_Number, Card_Number_Set,
                        Account_Number_Set -> Client
          Get_Client_Number : Client       -> Client_Number
     eqns forall n: Client_Number, k: Card_Number_Set, a: Account_Number_Set
       ofsort Client_Number
         Get_Client_Number(Make_Client(n, k, a)) = n;
```

```
  endtype

  type Card_Type is Card_Number_Set_Type, Client_Number_Set_Type,
                    Account_Number_Set_Type
     sorts Card
     opns Make_Card       : Card_Number, Client_Number,
                             Account_Number -> Card
          Get_Card_Number : Card           -> Card_Number
     eqns forall n: Card_Number, c: Client_Number, a: Account_Number
       ofsort Card_Number
         Get_Card_Number(Make_Card(n, c, a)) = n;
  endtype
(*----------------------------------------------------------------------
```

In our specification the objects `Client` and `Card` play a less important role, and so they are only specified as abstract data types. If we decided to increase their importance in the system they would need a process as well.

Each service name, which is part of the message sent to an object is defined here.

```
----------------------------------------------------------------------*)
  type Op_Names is
    sorts Op_Name
    opns
      deposit, withdraw, withdraw_cash, balance, open_account, close_account,
      create, remove, print_mini_statement, deposit_cash, rtn_withdraw,
      rtn_withdraw_cash, rtn_withdraw_cheque, set_mini_statement,
      mini_statement_details, rtn_print, rtn_deposit_cheque,
      rtn_deposit_cash, deposit_cheque, enquire_cheque, give_enquire_cheque,
      ask_transfer, withdraw_cheque, rtn_open_account, send_cheque_to_withdraw,
      receive_cheque_from_withdraw, create_cheque, set_standing_order,
      cancel_standing_order, receive_transfer, send_transfer, transfer_credit,
      cheque_debited, so_create, so_cancel, full_deposit, ask_withdraw_cheque,
      receive_cheque_balance, perhaps_deposit, rtn_balance, transfer_debit,
      ch_create, withdraw_others_bank_cheque, send_cheque_balance,
      rtn_set_standing_order : -> Op_Name
  endtype
(*----------------------------------------------------------------------
```

### Behaviour

The top level behaviour expression is directly created from the OCD represented in Figure 2.19 and by following the algorithm presented in [3].

```
----------------------------------------------------------------------*)
behaviour
  (hide ob2, cs, ba in
    ( Other_Banks[ob1, ob2, ba](Insert(This_Bank, {} of Bank_Name_Set))
     |[ob2]|
      ( Tellers[t, ob2, cs, ba]
      |[cs]|
        Financial_Instruments[ob2, cs, ba]
      )
    )
   |[ba]|
    Bank_Account[ba]
  )
 |[t, ob1]|
  Interface_Scenario[t, ob1]
 where
```

```
(*-----------------------------------------------------------------------
```

As shown in Figure 2.17, the object classes `Automatic Teller` and `Counter Teller` are subclasses of the Superclass `Entry Station`. The process `Entry Stations` below groups both the `Automatic Tellers` and the `Counter Tellers` in a single process.

```
-----------------------------------------------------------------------*)
    process Tellers [t, ob, a, c] : noexit :=
      ( Automatic_Tellers[t, c]({} of Id_Tellers_Set)
      |||
        Counter_Tellers[t, ob, a, c]({} of Id_Tellers_Set)
      )
    where
(*-----------------------------------------------------------------------
```

The superclass `Entry Station` defines the service `withdraw cash` which is offered by both kinds of tellers.

```
-----------------------------------------------------------------------*)
      process Entry_Station[t, c](id: Id_Tellers) : exit(Id_Tellers):=
        t !withdraw_cash !id ?n: Account_Number ?m: Money;
        c !withdraw !n !m;
        c !rtn_withdraw !n ?valid: Bool;
        t !rtn_withdraw_cash !id !valid;
        exit(id)
      endproc

   process Automatic_Tellers[t, c](atms: Id_Tellers_Set) : noexit :=
     t !create ?id: Id_Tellers [(id notin atms) and Is_ATM(id)];
      ( Automatic_Teller[t, c](id)
     |||
       Automatic_Tellers[t, c](Insert(id, atms))
      )
    where
     process Automatic_Teller[t, c](id: Id_Tellers) : noexit :=
        ( Entry_Station[t, c](id)
        []
          t !print_mini_statement !id ?n: Account_Number
              [Is_Cheque_Acc(n)];
          c  !set_mini_statement !n ?a: State_Account;
          t !rtn_print !id !n !a;
          exit(id)
       ) >> accept id: Id_Tellers in Automatic_Teller[t, c](id)
     endproc
   endproc  (* end Automatic_Tellers *)
(*-----------------------------------------------------------------------
```

The technique we are using to create a new automatic teller identifier

```
   t !create ?id: Id_Tellers [(id notin atms) and Is_ATM(id)];
```

uses the selection predicate `[(id notin atms) and Is ATM(id)]` which guarantees that the identifier generated does not exist yet and that its value belongs to the range defined by the boolean function `Is ATM(id)`. The extended action denotation `?id: Id Tellers` specifies the set of values allowed for `id`.

As the SMILE simulator offers value generation, we will use it to get new identifiers for automatic tellers. Thus, in the `Interface Scenario`, we will have an event synchronization of the type:

```
t !create ?id: Id_Tellers;
```

By using this technique we avoid having to specify how to create the identifiers. The same technique will be used for Counter Tellers and Other Banks.

Counter Tellers and Other Banks will be specified in the same way as Automatic Tellers.

```
---------------------------------------------------------------------*)
   process Counter_Tellers[t, ob, a, c](cts: Id_Tellers_Set) : noexit :=
     t !create ?id: Id_Tellers [(id notin cts) and Is_CT(id)];
      ( Counter_Teller[t, ob, a, c](id)
     |||
        Counter_Tellers[t, ob, a, c](Insert(id, cts))
      )
    where
     process Counter_Teller[t, ob, a, c](id: Id_Tellers) : noexit :=
       (  Entry_Station[t, c](id)
        []
          t !open_account !id ?ta: Kind_Account;
          c !create !ta ?n: Account_Number;
          t !rtn_open_account !id !n;
          exit(id)
        []
          t !close_account !id ?n: Account_Number;
          c !remove !n;
          exit(id)
        []
          t !deposit_cash !id ?n: Account_Number ?m: Money;
          c !deposit !n !m;
          t !rtn_deposit_cash !id !n !m;
          exit(id)
        []
          t !deposit_cheque !id ?bk: Bank_Name
            ?ch_from: Account_Number ?ch_to: Account_Number
            ?ch: Cheque_Number ?m: Money [is_Cheque_Acc(ch_from)];
          a !create_cheque !ch !bk !ch_from !ch_to !m;
          t !rtn_deposit_cheque !id !ch_to !m;
          exit(id)
        []
          t !withdraw_cheque !id ?bk: Bank_Name ?n: Account_Number
            ?ch: Cheque_Number ?m: Money
            [(bk eq This_Bank) and Is_Cheque_Acc(n)];
             (* client withdraws cheque in his/her bank *)
          a !create_cheque !ch !bk !n !m;
             (* process cheque is asked to withdraw cheque *)
          a !cheque_debited !ch !bk !n ?cheque_valid: Bool;
              (* Cheque sends information about whether or not the
                 withdrawl was successful *)
          t !rtn_withdraw_cheque !id !n !cheque_valid;
              (* teller gives money and receipt to the client *)
          exit(id)
        []
          t !enquire_cheque !id ?ch: Cheque_Number;
          a !enquire_cheque !ch ?this_cheque: State_Cheque;
          t !give_enquire_cheque !id !this_cheque;
          exit(id)
        []
          t !balance !id ?n:Account_number;
          c !balance !n ?b: Balance;   (* balance value *)
          t !rtn_balance !id !n !b;
```

```
          exit(id)
        []
          t !ask_transfer !id ?from_acc: Account_Number
            ?to_acc: Account_Number ?bk: Bank_Name ?m: Money;
                                   (* asked by the clients *)
          c !withdraw !from_acc !m;
          c !rtn_withdraw !from_acc ?valid: Bool;
          ( [valid] ->
             (
               [bk eq This_bank] -> c !deposit !to_acc !m; exit(id)
              []
               [bk ne This_Bank] -> ob !send_transfer !bk !to_acc !m;
                                     exit(id)  (* to other banks *)
             )
           []
            [not(valid)] -> exit(id)
          )
        []
          t !set_standing_order !id ?from_acc: Account_Number
            ?to_acc: Account_Number ?bk: Bank_Name ?m: Money
            [Is_Cheque_Acc(from_acc) and Is_Cheque_Acc(to_acc)];
          a !so_create !from_acc !to_acc !bk !m
             ?so_number: SO_Number;
          t !rtn_set_standing_order !id !so_number;
          exit(id)
        []
          t !id !cancel_standing_order ?so_number: SO_Number;
          a !so_cancel !so_number;
          exit(id)
       ) >> accept id: Id_Tellers in Counter_Teller[t, ob, a, c](id)
     endproc
   endproc (* end Counter_Tellers *)
 endproc (* end Tellers *)
(*----------------------------------------------------------------------
```

Each instance of Other Bank only deals with transactions referring to a given bank. Hence, the number of objects created is equal to the number of banks we deal with.

Notice that the identifier of our bank is inserted as an element of the set of banks, when Other_Banks was called initially.

```
----------------------------------------------------------------------*)
   process Other_Banks[ob, a, c](bks: Bank_Name_Set) : noexit :=
     ob !create ?bk: Bank_Name [bk notin bks];
      ( Other_Bank[ob, a, c](bk)
     |||
        Other_Banks[ob, a, c](Insert(bk, bks))
      )
   where
     process Other_Bank[ob, a, c](bk: Bank_Name) : noexit :=
       ( ob !receive_transfer !bk ?n: Account_Number ?m: Money;
          c  !deposit !n !m;
          exit(bk)
        []
          a  !send_transfer !bk ?n: Account_Number ?m: Money;
               (* either from counter teller or standing order *)
          ob !send_transfer !bk !n !m;
          exit(bk)
        []
          a  !ask_withdraw_cheque !bk ?n: Account_Number
```

```
            ?ch: Cheque_Number ?m: Money [Is_Cheque_Acc(n)];
              (* from cheque *)
          ob !send_cheque_to_withdraw !bk !n !ch !m;
          exit(bk)
        []
          ob !receive_cheque_balance !bk ?n: Account_Number
            ?ch: Cheque_Number ?m: Money ?valid_result: Bool;
              (* confirmation of ask_withdraw_cheque *)
          a  !receive_cheque_balance !n !ch !m !valid_result;
              (* goes to cheque *)
          exit(bk)
        []
          ob !receive_cheque_from_withdraw !bk ?n: Account_Number
            ?ch: Cheque_Number ?m: Money;
              (* another bank asks to withdraw a cheque *)
          c  !withdraw !n !m;
          c  !rtn_withdraw !n ?valid_result: Bool;
          ob !send_cheque_balance !bk !n !ch !m !valid_result;
          exit(bk)
      ) >> accept bk: Bank_Name in Other_Bank[ob, a, c](bk)
    endproc
  endproc (* end Other_Banks *)
(*---------------------------------------------------------------------
```

The process `Financial Instruments` is a composite object, and the components are the class templates `Cheque` and `Standing Order` (see Figure 2.19). (First we model the object generators for these two class templates.)

```
---------------------------------------------------------------------*)
  process Financial_Instruments[ob, a, c] : noexit :=
    ( Cheques[ob, a, c]
    |||
      Standing_Orders[ob, a, c]({} of SO_Number_Set)
    )

   where
(*---------------------------------------------------------------------
```

The `Cheques` generator does not hold the cheque identifiers, as they are provided by the outside world.

`Cheque` can be specified in two different ways: first, we could use a boolean flag which, when "true", only allows a deposit or a withdrawal to be accepted. After a deposit or a withdrawal occurs, the flag will be set to "false" and only state information enquiries are then allowed. One alternative is to split the cheque process into two processes, one dealing with the active part of the cheques (deposit and withdraw services), and other dealing with the passive part (enquiries about the cheque state information). The latter is adopted.

As when a cheque is created it is to be used for withdrawal or deposit we opted for a solution where we have two different parts: one created for the service withdrawal and the other for the service deposit.

```
---------------------------------------------------------------------*)
    process Cheques[ob, a, c] : noexit :=
      a !create_cheque ?ch: Cheque_Number ?bk: Bank_Name
        ?from_acc: Account_Number ?to_acc: Account_Number ?m: Money;
      ( Cheque_Deposit[ob, a, c](Make_Cheque(ch, from_acc, to_acc, bk, m))
      |||
        Cheques[ob, a, c]
      )
```

```
 []
  a !create_cheque ?ch: Cheque_Number ?bk: Bank_Name
    ?n: Account_Number ?m: Money;
  ( Cheque_Withdraw[ob, a, c](Make_Cheque(ch, n, n, bk, m))
  |||
    Cheques[ob, a, c]
  )
where

  process Cheque_Deposit[ob, a, c](this_cheque: State_Cheque) : noexit :=
  ( Cheque_Deposit_1[ob, a, c](this_cheque)
   >> accept upd_cheque: State_Cheque in
    Passive_Cheque_Deposit[a](upd_cheque)
  )
  where
    process Cheque_Deposit_1[ob, a, c](this_cheque: State_Cheque) :
             exit(State_Cheque) :=
      c  !perhaps_deposit !Get_Acc_To(this_cheque)
         !Get_Amount(this_cheque);
       ( [Get_Bank(this_cheque) eq This_Bank] ->
         ( c !withdraw !Get_Acc_From(this_cheque)
             !Get_Amount(this_cheque);
           c !rtn_withdraw !Get_Acc_From(this_cheque) ?cheque_valid: Bool;
           c !full_deposit !Get_Acc_To(this_cheque)
             !Get_Amount(this_cheque) !cheque_valid;
           exit(this_cheque)
          )
       []
         [Get_Bank(this_cheque) ne This_Bank] ->
         (ob !ask_withdraw_cheque !Get_Bank(this_cheque)
             !Get_Acc_From(this_cheque)
             !Get_Cheque_Number(this_cheque) !Get_Amount(this_cheque);
                                          (* to other banks *)
          ob !receive_cheque_balance !Get_Acc_From(this_cheque)
             !Get_Cheque_Number(this_cheque) !Get_Amount(this_cheque)
             ?valid_result: Bool;
           c !full_deposit !Get_Acc_To(this_cheque)
             !Get_Amount(this_cheque) !valid_result;
           exit(this_cheque)
         )
       )
    endproc (* Cheque_Deposit_1 *)

    process Passive_Cheque_Deposit[a](this_cheque: State_Cheque) : noexit :=
      ( a !enquire_cheque !Get_Cheque_Number(this_cheque) !this_cheque;
        exit(this_cheque) (* cheque state information *)
      ) >> accept upd_cheque: State_Cheque in
       Passive_Cheque_Deposit[a](upd_cheque)
    endproc  (* Passive_Cheque_Deposit *)

  endproc (* Cheque_Deposit *)
(*------------------------------------------------------------------------
```

As a deposit cheque service can take several days to be accomplished (it may be necessary to ask another bank if the account corresponding to the cheque has funds) we have a `perhaps deposit` event. A `perhaps deposit` event adds to the pending balance the amount to be deposited, and only when we know that the cheque has funds do we credit the account, by doing a `full deposit`.

Notice that we cannot access the passive part of cheque until the "active" part of cheque is finished. This sequentiality is introduced by the enable operator ">>".

The same scheme is followed when the account corresponding to a cheque to withdraw money belongs to the same bank as the one where it is issued.

```
----------------------------------------------------------------------*)
       process Cheque_Withdraw[ob, a, c](this_cheque: State_Cheque) : noexit :=
       ( Cheque_Withdraw_1[ob, a, c](this_cheque)
        >> accept upd_cheque: State_Cheque in
         Passive_Cheque_Withdraw[a](upd_cheque)
       )
       where
         process Cheque_Withdraw_1[ob, a, c](this_cheque: State_Cheque) :
                   exit(State_Cheque) :=
           c !withdraw !Get_Acc_From(this_cheque) !Get_Amount(this_cheque);
           c !rtn_withdraw !Get_Acc_From(this_cheque) ?cheque_valid: Bool;
          ob !cheque_debited !Get_Cheque_Number(this_cheque)
             !Get_Bank(this_cheque) !Get_Acc_From(this_cheque)
             !cheque_valid;
           exit(this_cheque)
         endproc  (* Cheque_Withdraw_1 *)

         process Passive_Cheque_Withdraw[a](this_cheque: State_Cheque) : noexit :=
           ( a !enquire_cheque !Get_Cheque_Number(this_cheque) !this_cheque;
             exit(this_cheque) (* cheque state information *)
           ) >> accept upd_cheque: State_Cheque in
            Passive_Cheque_Withdraw[a](upd_cheque)
         endproc (* Passive_Cheque_Withdraw *)

       endproc  (* Cheque_Withdraw *)
     endproc  (* Cheques *)
(*----------------------------------------------------------------------
```

The main service in `Standing Order` is to withdraw a certain amount from the client account and eventually to deposit the same amount in another account, or to send a transfer to another bank.

Because this service must be accomplished on fixed days, it is defined as an internal event, initiated by the internal event `appropriate date`.

Once more, a standing order number needs to be generated before the object is created. This is accomplished by:

```
   a !so_create ?n1: Account_Number ?n2: Account_Number
     ?bk: Bank_Name ?m: Money ?so_counter: SO_Number [so_counter notin sos]
```

This creates a new standing order with a new number together with the information received from a counter teller.

```
----------------------------------------------------------------------*)
     process Standing_Orders[ob, a, c](sos: SO_Number_Set) : noexit :=
       a !so_create ?n1: Account_Number ?n2: Account_Number
         ?bk: Bank_Name ?m: Money ?so_counter: SO_Number [so_counter notin sos];
       ( Standing_Order[ob, a, c](Make_SO(so_counter, n1, n2, bk, m))
       |||
         Standing_Orders[ob, a, c](Insert(so_counter, sos))
       )
     where
       process Standing_Order[ob, a, c](this_so: State_Standing_Order) : noexit :=
         ( a !so_cancel !Get_SO_Number(this_so); stop
         []
           (hide appropriate_date in
```

```
          (appropriate_date;
           c !withdraw !Get_Paying_Client(this_so)
             !Get_Payable_Amount(this_so);
           c !rtn_withdraw !Get_Paying_Client(this_so) ?has_funds: Bool;
           ( [has_funds] ->
              ([ Get_Bank_Name(this_so) eq This_Bank] ->
                   (c !deposit !Get_Receiving_Client(this_so)
                      !Get_Payable_Amount(this_so);
                    exit(this_so)
                   )
               []
                [Get_Bank_Name(this_so) ne This_Bank] ->
                   (ob !send_transfer !Get_Bank_Name(this_so)
                       !Get_Receiving_Client(this_so)
                       !Get_Payable_Amount(this_so);
                    exit(this_so)
                   )
              )
            []
             [not(has_funds)] -> exit(this_so) (* error condition *)
           )
          )
        )
      ) >> accept upd_so: State_Standing_Order in
       Standing_Order[ob, a, c](upd_so)
    endproc
  endproc  (* end Standing_Order *)

endproc  (* end Financial_Instruments *)
(*----------------------------------------------------------------------
```

As we can see, looking at Figure 2.17, Savings Account and Cheque Account are inherited subclasses of the superclass Account. The processes Savings Account and Cheque Account are pure extensions of the superclass Superclass Account.

The choice operator, *choice*, allows us to introduce non-determinism.

```
----------------------------------------------------------------------*)
     process Account[c](this_account: State_Account): exit(State_Account) :=
       c !deposit !Get_Account_Number(this_account) ?m: Money;
       exit(Credit_Account(this_account, m))
     []
       c !withdraw !Get_Account_Number(this_Account) ?m: Money;
       ( choice if_money: Bool []
         [if_money] -> c !rtn_withdraw !Get_Account_Number(this_account) !true;
                       exit(Debit_Account(this_account, m))
       []
         [not (if_money)] -> c !rtn_withdraw
                               !Get_Account_Number(this_account) !false;
                             exit(this_account)
       )
     []
       c !balance !Get_Account_Number(this_account)
         !Get_Balance(this_account);
       exit(this_account)
     []
       c !perhaps_deposit !Get_Account_Number(this_account) ?m: Money;
       exit(Credit_Pending(this_account, m))
     []
       c !full_deposit !Get_Account_Number(this_account) ?m: Money ?valid: Bool;
```

```
        ( [valid] -> exit(Add_Credit_Pending(this_account, m))
        []
          [not (valid)] -> exit(Sub_Credit_Pending(this_account, m))
        )
      endproc
(*----------------------------------------------------------------------
```

Several programming languages, such as Eiffel, offer a `create` service for each subclass. In a first approach we thought we could define it in the superclass and then let it be inherited for each subclass. However, when the abstract data types were introduced we realised that this solution would not work.

We want account numbers of both types of accounts to belong to the same sort. The solution is, as for the identifiers of automatic tellers and counter tellers, to share a set of values in which each kind of account would deal with different ranges of values. With this option, the simplest way to specify the create service, and then deal with the resulting value is to define `create` in the process that generates the instances of each subclass.

If we had decided to define two sorts of account numbers, one for each kind of account, inheritance would be difficult to specify.

In order to protect each account from being accessed by more than one process at a time while allowing different accounts to be accessed concurrently, semaphores, monitors, or another kind of data protection techniques can be used. The way we specify the accounts (where each account is a process by itself), does not require the extra information for data protection. Due to the nature of LOTOS, it is not difficult to specify the system as a set of concurrent objects (which is one of the goals of the object-oriented view), giving us for free the data protection we require.

```
----------------------------------------------------------------------*)
   process Bank_Account[c] : noexit :=
        ( Savings_Accounts[c]({} of Account_Number_Set)
        |||
          Cheque_Accounts[c]({} of Account_Number_Set)
        )
      where
(*----------------------------------------------------------------------
```

The behaviour expression

```
   Savings_Accounts[c]({} of Account_Number_Set)
```

represents a process that encapsulates a set of processes each of which encapsulates a savings account. (The same for `Cheque Accounts`). It gives us a parallel solution, where each account is a process. However we could have chosen `Savings Accounts` to encapsulate a set of accounts, and this would be a sequential solution. From the outside point of view, we can replace one solution by the other without any side effect.

```
----------------------------------------------------------------------*)
      process Savings_Accounts[c](accs: Account_Number_Set) : noexit :=
        c !create !saving ?acc_counter: Account_Number
           [(acc_counter notin accs) and Is_Savings_Acc(acc_counter)];
        ( Savings_Account[c](Make_Account(acc_counter, Some_Balance),
                                  This_Prd, This_Inter)
        |||
          Savings_Accounts[c](Insert(acc_counter, accs))
        )
        where
          process Savings_Account[c](this_account: State_Account,
                  prd: Period, int_rate: Interest) : noexit :=
            ( hide credit_interest, update_date in
               ( ( Account[c](this_account)
```

```
                   >> accept upd_account: State_Account in
                  exit(upd_account, prd, int_rate)
                )
               []
                 credit_interest;
                 exit(Credit_Interest(this_account, This_Mon), prd, int_rate)
               []
                 update_date;
                 exit(this_account, Update_Date(prd), int_rate)
               []
                 c !remove !Get_Account_Number(this_account); stop
              )
          ) >> accept upd_account: State_Account,
                      upd_prd: Period, upd_int_rate: Interest in
           Savings_Account[c](upd_account, upd_prd, upd_int_rate)
        endproc
     endproc
(*----------------------------------------------------------------------
```

New account numbers are given by using value generation. The behaviour expression

```
   c !create !saving ?acc_counter: Account_Number
     [(acc_counter notin accs) and Is_Savings_Acc(acc_counter)];
```

generates the value acc_counter and passes it as the new account number when it synchronises with Counter_Teller in the behaviour expression

```
   c !create !ta ?n: Account_Number;
```

```
----------------------------------------------------------------------*)
     process Cheque_Accounts[c](accs: Account_Number_Set): noexit :=
       c !create !chequing ?acc_counter: Account_Number
          [(acc_counter notin accs) and Is_Cheque_Acc(acc_counter)];
       ( Cheque_Account[c](Make_Account(acc_counter, Some_Balance),
                            {} of Card_Number_Set, {} of SO_Number_Set)
       |||
         Cheque_Accounts[c](Insert(acc_counter,accs))
       )
       where
         process Cheque_Account[c](this_account: State_Account,
                   cards: Card_Number_Set, sos: SO_Number_Set) : noexit :=
          ( ( Account[c](this_account)
                >> accept upd_account: State_Account in
               exit(upd_account, cards, sos)
             )
            []
              c !set_mini_statement !Get_Account_Number(this_account)
                !this_account;
              exit(this_account, cards, sos)
            []
              c !remove !Get_Account_Number(this_account); stop
          ) >> accept upd_account: State_Account, upd_cards: Card_Number_Set,
                      upd_sos: SO_Number_Set in
           Cheque_Account[c](upd_account, upd_cards, upd_sos)
         endproc
     endproc

   endproc (* end Bank_Account *)
(*----------------------------------------------------------------------
```

The two parameters `cards` and `sos` correspond to the relationships between `Cheque Account` and `Card` and between `Cheque Account` and `Standing Order`.

The class templates `client` and `card` in the object model depicted in Figure 2.17 play a minor role in our system. This situation could be changed if the behaviour of the system was extended to incorporate other properties. In the present solution, both class templates are defined as abstract data types. The record of the clients and the correspondant cards, are kept in a set of clients and a set of cards. A new element in each set is inserted when a new account is created (with the necessary checks to see if the client already exists).

### Validation

Interface scenarios are important to help the validation of the specification. The `Interface Scenario` process acts as a management process which creates the automatic tellers, counter tellers and other banks objects before making use of their services.

We are using value generation in order to work with symbols instead of values. The SMILE simulator [6] allows symbolic execution of a specification where a set of possible values is used rather than particular values. Many more behaviours can then be examined with each simulation than is possible when all data values have to be instantiated. SMILE uses a narrowing algorithm to determine when a combination of conditions can never be true.

In `Interface Scenario` the automatic tellers, counter tellers and accounts are created by value generation. Thus, the result is a symbol in the set defined for each of those objects.

```
----------------------------------------------------------------------*)
    process Interface_Scenario[t, ob]: noexit :=
     (* create an automatic teller and a counter teller *)
       t !create ?idc: Id_Tellers [Is_CT(idc)];
       t !create ?ida: Id_Tellers [Is_ATM(ida)];

     (* open a chequing account from counter teller *)
       t !open_account !idc !chequing;
       t !rtn_open_account !idc ?nc1: Account_Number [Is_Cheque_Acc(nc1)];

     (* open another chequing account from counter teller *)
       t !open_account !idc !chequing;
       t !rtn_open_account !idc ?nc2: Account_Number [Is_Cheque_Acc(nc2)];

     (* open a saving account from counter teller  *)
       t !open_account !idc !saving;
       t !rtn_open_account !idc ?ns: Account_Number [Is_Savings_Acc(ns)];

     (* deposit cheque from our bank*)
       t !deposit_cheque !idc !This_Bank !nc1 !nc2
          ?ch: Cheque_Number !This_Mon;
       t !rtn_deposit_cheque !idc !nc2 !This_Mon;

     (* balance from automatic teller*)
       t !print_mini_statement !ida !nc1;
       t !rtn_print !ida !nc1 ?a: State_Account;

     (* withdrawal from automatic teller *)
       t !withdraw_cash !ida !nc1 !This_Mon;
       t !rtn_withdraw_cash !ida ?val: Bool;

     (* deposit from counter teller *)
       t !deposit_cash !idc !nc1 !This_Mon
          [Is_Cheque_Acc(nc1) and Is_CT(idc)];
       t !rtn_deposit_cash !idc !nc1 !This_Mon;
```

```
      (* balance from counter teller *)
        t !balance !idc !ns;
        t !rtn_balance !idc !ns ?b: Balance;

       (hide success in success; stop)
   endproc (* Interface Scenario *)

endspec
(*---------------------------------------------------------------------------
```

# Appendix B

# LOTOS specification for Warehouse management system

```
specification Warehouse [usr]: noexit

  library
    Boolean, NaturalNumber, Set
  endlib

  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
         h   : Id -> Nat
         _eq_, _ne_, _lt_ : Id, Id -> Bool
         First_Set  : Id -> Bool
         Second_Set : Id -> Bool
         Third_Set  : Id -> Bool
         Fourth_Set : Id -> Bool
    eqns forall n1, n2: Id
      ofsort Nat
        h(id1) = 0;
        h(id2) = succ(h(id1));
        h(id3) = succ(h(id2));
        h(id4) = succ(h(id3));
        h(id5) = succ(h(id4));
        h(id6) = succ(h(id5));
        h(id7) = succ(h(id6));
        h(id8) = succ(h(id7));
        h(id9) = succ(h(id8));
        h(id10) = succ(h(id9));
        h(id12) = succ(h(id11));
        h(id13) = succ(h(id12));
        h(id14) = succ(h(id13));
        h(id15) = succ(h(id14));
        h(id16) = succ(h(id15));
        h(id17) = succ(h(id16));
        h(id18) = succ(h(id17));
        h(id19) = succ(h(id18));
        h(id20) = succ(h(id19));
```

```
    ofsort Bool
      n1 eq n2 = h(n1) eq h(n2);
      n1 ne n2 = h(n1) ne h(n2);
      n1 lt n2 = h(n1) lt h(n2);
      First_Set(n1)  = h(n1) lt h(id5);
      Second_Set(n1) = not(h(n1) lt h(id5))  and (h(n1) lt h(id10));
      Third_Set(n1)  = not(h(n1) lt h(id10)) and (h(n1) lt h(id15));
      Fourth_Set(n1) = not(h(n1) lt h(id15)) and (h(n1) lt h(id20));
endtype

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

type Warehouse_Id_Type is Id_Type
   renamedby
   sortnames Warehouse_Id for Id
endtype

type Local_Plan_Id_Type is Id_Type
   renamedby
   sortnames Local_Plan_Id for Id
endtype

type Network_Id_Type is Id_Type
   renamedby
   sortnames Network_Id for Id
endtype

type Truck_Id_Type is Id_Type
   renamedby
   sortnames Truck_Id for Id
endtype

type Plan_Id_Type is Id_Type
   renamedby
   sortnames Plan_Id for Id
endtype

type Office_Id_Type is Id_Type
   renamedby
   sortnames Office_Id for Id
endtype

type Red_Req_Id_Set_Type is Set_Id_Type
   renamedby
   sortnames Red_Req_Id for Id
             Red_Req_Id_Set for Set
endtype

type Ins_Req_Id_Set_Type is Set_Id_Type
   renamedby
   sortnames Ins_Req_Id for Id
             Ins_Req_Id_Set for Set
endtype

type Rem_Req_Id_Set_Type is Set_Id_Type
```

```
   renamedby
   sortnames Rem_Req_Id for Id
             Rem_Req_Id_Set for Set
endtype

type Item_Id_Set_Type is Set_Id_Type
   renamedby
   sortnames Item_Id for Id
             Item_Id_Set for Set
endtype

type Item_Type is Item_Id_Set_Type, Item_Place_Id_Set_Type,
                  Warehouse_Id_Type, Boolean
   sorts Item_State
   opns Make_Item : Item_Id, Warehouse_Id, Item_Place_Id, Bool, Bool
                        -> Item_State
        Get_It_Id : Item_State      -> Item_Id
        Get_Ware : Item_State -> Warehouse_Id
        Get_Place_Id :Item_State -> Item_Place_Id
        Can_Move :Item_State -> Bool
        Move : Warehouse_Id, Item_Place_Id, Item_State -> Item_State
        Mark_m : Item_State -> Item_State
        Mark_r : Item_State -> Item_State
   eqns forall it: Item_Id, ip, nip: Item_Place_Id, w, nw: Warehouse_Id,
               s1, s2: Bool
     ofsort Item_Id
       Get_It_Id(Make_Item(it, w, ip, s1, s2)) = it;
     ofsort Item_Place_Id
       Get_Place_Id(Make_Item(it, w, ip, s1, s2)) = ip;
     ofsort Warehouse_Id
       Get_Ware(Make_Item(it, w, ip, s1, s2)) = w;
     ofsort Item_State
        Move(nw, nip, Make_Item(it, w, ip, s1, s2)) = Make_Item(it, nw, nip, true, s2);
        Mark_m(Make_Item(it, w, ip, s1, s2)) = Make_Item(it, w, ip, false, s2);
        Mark_r(Make_Item(it, w, ip, s1, s2)) = Make_Item(it, w, ip, s1, false);
     ofsort Bool
        Can_Move(Make_Item(it, w, ip, s1, s2)) = s1 and s2;
 endtype

type Item_Place_Id_Set_Type is Set_Id_Type
   renamedby
   sortnames Item_Place_Id for Id
             Item_Place_Id_Set for Set
endtype

type Item_Place_Type is Item_Place_Id_Set_Type, Item_Id_Set_Type, Boolean
   sorts Item_Place
   opns Make_Place   : Item_Place_Id, Item_Id -> Item_Place
        Get_Place_Id: Item_Place -> Item_Place_Id
        Get_It_Id : Item_Place -> Item_Id
   eqns forall ip: Item_Place_Id, it: Item_Id
     ofsort Item_Place_Id
       Get_Place_Id(Make_Place(ip,it)) = ip;
     ofsort Item_Id
       Get_It_Id(Make_Place(ip,it)) = it;
 endtype

type Rem_Req_Type is Rem_Req_Id_Set_Type, Item_Id_Set_Type,
```

```
                      Time_Type, Warehouse_Id_Type, Boolean
   sorts Rem_Req
   opns Make_Rem_Req   : Rem_Req_Id, Item_Id, Warehouse_Id, Time
                           -> Rem_Req
        Get_It_Id: Rem_Req -> Item_Id
        Get_Rem_Id: Rem_Req -> Rem_Req_Id
        Get_Ware: Rem_Req -> Warehouse_Id
        Get_Time: Rem_Req -> Time
   eqns forall ir: Rem_Req_Id, it: Item_Id, dest: Warehouse_Id, t: Time
     ofsort Item_Id
       Get_It_Id(Make_Rem_Req(ir, it, dest, t)) = it;
     ofsort Rem_Req_Id
       Get_Rem_Id(Make_Rem_Req(ir, it, dest, t)) = ir;
     ofsort Warehouse_Id
       Get_Ware(Make_Rem_Req(ir, it, dest, t)) = dest;
     ofsort Time
       Get_Time(Make_Rem_Req(ir, it, dest, t)) = t;
 endtype

type Red_Req_Type is Red_Req_Id_Set_Type, Item_Id_Set_Type, Boolean,
                      Warehouse_Id_Type, Time_Type
   sorts Red_Req
   opns Make_Red_Req   : Red_Req_Id, Item_Id, Warehouse_Id,
                         Warehouse_Id, Time -> Red_Req
        Get_It_Id: Red_Req -> Item_Id
        Get_Red_Id: Red_Req -> Red_Req_Id
        Get_From: Red_Req -> Warehouse_Id
        Get_Dest: Red_Req -> Warehouse_Id
        Get_Time: Red_Req -> Time
   eqns forall ir: Red_Req_Id, it: Item_Id, from, dest: Warehouse_Id, t: Time
     ofsort Item_Id
       Get_It_Id(Make_Red_Req(ir, it, from, dest, t)) = it;
     ofsort Red_Req_Id
       Get_Red_Id(Make_Red_Req(ir, it, from, dest, t)) = ir;
     ofsort Warehouse_Id
       Get_From(Make_Red_Req(ir, it, from, dest, t)) = from;
       Get_Dest(Make_Red_Req(ir, it, from, dest, t)) = dest;
     ofsort Time
       Get_Time(Make_Red_Req(ir, it, from, dest, t)) = t;
 endtype

type Ins_Req_Type is Ins_Req_Id_Set_Type, Item_Id_Set_Type, Boolean,
                      Warehouse_Id_Type, Time_Type
   sorts Ins_Req
   opns Make_Ins_Req   : Ins_Req_Id, Time, Warehouse_Id -> Ins_Req
        Get_Ins_Id: Ins_Req -> Ins_Req_Id
        Get_Time: Ins_Req -> Time
        Get_Ware_Id:Ins_Req -> Warehouse_Id
   eqns forall ir: Ins_Req_Id, t: Time, w: Warehouse_Id
     ofsort Ins_Req_Id
       Get_Ins_Id(Make_Ins_Req(ir, t, w)) = ir;
     ofsort Time
       Get_Time(Make_Ins_Req(ir, t, w)) = t;
     ofsort Warehouse_Id
       Get_Ware_Id(Make_Ins_Req(ir, t, w)) = w;
 endtype

type Time_Type is
```

```
      sorts Time
      opns This_Time: -> Time
   endtype

   type Events is
     sorts Event
     opns
       create_ins_req, check_expected, confirm_inserted, rtn_confirm_inserted,
       create, check_item, create_red_req, create_rem_req, add_redis_req,
       get_trans_req, choose_red, request_eta, rtn_request_eta, fetch_items,
       rtn_fetch_items, return_space,
       confirm_loaded, get_eta, rtn_get_eta, warn_warehouse, warn_arrival,
       is_delivered, rtn_is_delivered, get_position, give_place_item,
       insert, notify_complete, insert_query, rtn_insert_query, add_rem_req,
       choose_rem, remove, rm_fetch_items, init_move_item, rtn_init_move_item,
       fetch, init_new_item, rtn_init_new_item, confirm_delivered,
       init_remove_item, rtn_init_remove_item, update_item, get_place,
       mark_move, mark_remove, reserve_space, rtn_reserve_space
       : -> Event
   endtype

behaviour
   (hide req1, req2, usr1, usr2, usr3 in
      ((hide it1, to_n, from_n in
       ( Office_Interf[usr3, req2, it1, to_n, from_n](id1 of Office_Id)
         |||
         Warehouse[usr1, req1, req2, it1, to_n, from_n](id1 of Warehouse_id)
         |||
         Warehouse[usr1, req1, req2, it1, to_n, from_n](id2 of Warehouse_id)
       )
       |[it1, to_n, from_n]|
       (Items[it1]({} of Item_Id_Set) ||| Network[to_n, from_n](id6 of Network_Id))
      )
      |[req1, req2]|
      (hide p1, p2, tr_r in
        Request[req1, req2, p1, p2, tr_r]
        |[p1, p2, tr_r]|
        (hide p_tr in
          Truck_Interf[usr2, tr_r, p_tr](id1 of Truck_Id)
          |[p_tr]|
          Planning[p1, p2, p_tr](id1 of Plan_Id, {} of Red_Req_Id_Set,
                     {} of Rem_Req_Id_Set)
        )
      )
    )
  |[usr1, usr2, usr3]|
   Interface_Scenario[usr1, usr2, usr3])

   where

   process Warehouse[usr1, req1, req2, it1, to_n, from_n](wid: Warehouse_id) : noexit :=
      hide wp, w_lp in
      Warehouse_Interf[usr1, req1, req2, it1, wp, w_lp, to_n, from_n](wid)
      |[wp, w_lp]|
      (Places[wp]
       |||
       Local_Plan[w_lp](id14 of Local_Plan_Id,
                  Insert(id7, Insert(id8, Insert(id9, {} of Item_Place_Id_Set))))
```

```
   )
 where

   process Warehouse_Interf[usr1, req1, req2, it1, wp, w_lp, to_n, from_n]
             (wid: Warehouse_id) : noexit :=
     (  usr1 !insert_query !wid ?ir:Ins_Req_Id ?t:Time;
        req2 !check_expected !ir ?ok:Bool !wid !t;
        usr1 !rtn_insert_query !wid !ok !ir;
        ( [ok] ->
               w_lp !get_position ?lp_id: Local_Plan_Id ?ip:Item_Place_Id;
               it1 !create ?it: Item_id !wid !ip;
               wp !insert !ip !it;
               usr1 !give_place_item !wid !it !ip;
               exit(wid)
          []
           [not(ok)] ->
               exit(wid)
        )
      []
        usr1 !init_move_item !wid ?it: Item_Id ?t:Time ?dest: Warehouse_Id;
        it1 !check_item !it ?ok:Bool ?w: Warehouse_Id;
        ([not(ok)] -> usr1 !rtn_init_move_item !wid !ok;
                      exit(wid)
        []
         [ok]  -> ([w ne dest] ->
                      to_n !reserve_space ?net:Network_Id !dest;
                      to_n !rtn_reserve_space !net !dest ?is_space:Bool;
                      ([is_space] ->
                           req2 !create_red_req ?id: Red_Req_Id !it !w !dest !t;
                           it1 !mark_move !it;
                           usr1 !rtn_init_move_item !wid !true;
                           exit(wid)
                      []
                       [not(is_space)] ->
                           usr1 !rtn_init_move_item !wid !false;
                           exit(wid)
                      )
                  []
                   [w eq dest] ->
                      usr1 !rtn_init_move_item !wid !false;
                      exit(wid)
                  )
        )
      []
        req1 !rm_fetch_items !wid ?it:Item_Id ?t: Time;
        it1 !get_place !it ?plac: Item_Place_Id;
        usr1 !fetch !wid !it !plac;
        wp !remove !plac !it;
        it1 !remove !it !plac;
        w_lp !return_space ?lp_id: Local_Plan_Id;
        exit(wid)
      []
        req1 !fetch_items !wid ?it:Item_Id ?t: Time;
        it1 !get_place !it ?plac: Item_Place_Id;
        usr1 !fetch !wid !it !plac;
        wp !remove !plac !it;
        w_lp !return_space ?lp_id: Local_Plan_Id;
        exit(wid)
```

```
      []
        req1 !warn_warehouse !wid ?t: Time;
        usr1 !warn_arrival !wid !t;
        exit(wid)
      []
        req1 !confirm_inserted !wid ?it:Item_Id;
        w_lp !get_position ?lp_id: Local_Plan_Id ?ip:Item_Place_Id;
        wp !insert !ip !it;
        it1 !update_item !it !wid !ip;
        usr1 !give_place_item !wid !it !ip;
        req1 !rtn_confirm_inserted !wid;
        exit(wid)
      []
        from_n !reserve_space !wid ?net:Network_Id;
        w_lp !reserve_space ?lp_id: Local_Plan_Id ?ok:Bool;
        from_n !rtn_reserve_space !wid !net !ok;
        exit(wid)
      ) >> accept id: Warehouse_Id
           in Warehouse_Interf[usr1, req1, req2, it1, wp, w_lp, to_n, from_n](id)
  endproc

  process Places[wp] : noexit :=
    wp !insert ?ip: Item_Place_Id ?it: Item_Id;
    ( Place[wp](Make_Place(ip, it))
     |||
      Places[wp]
    )

  where
    process Place[wp] (this_itpl: Item_Place) : noexit :=
        wp !remove !Get_Place_Id(this_itpl) !Get_It_Id(this_itpl);
        stop
    endproc (*Place *)
  endproc (*Places *)

  process Local_Plan[w_lp](l_plan_id: Local_Plan_id,
                           ids: Item_Place_Id_Set) : noexit :=
       w_lp !get_position !l_plan_id ?ip: Item_Place_Id [ip isin ids];
       Local_Plan[w_lp](l_plan_id, ids)
    []
       w_lp !reserve_space !l_plan_id ?ok: Bool;
       Local_Plan[w_lp](l_plan_id, ids)
    []
       w_lp !return_space !l_plan_id;
       Local_Plan[w_lp](l_plan_id, ids)
  endproc (*Local_Plan*)

endproc (*Warehouse*)

process Items[it1](ids: Item_Id_Set) : noexit :=
   it1 !create ?id: Item_Id ?w: Warehouse_Id ?ip: Item_Place_Id
          [(id notin ids)];
   ( Item[it1](Make_Item(id, w, ip, true, true))
    |||
     Items[it1](Insert(id, ids))
   )
  []
    it1 !check_item ?id: Item_Id !false ?w: Warehouse_Id [(id notin ids)];
```

```
   Items[it1](ids)

where

process Item[it1](it: Item_State) : noexit :=
     it1 !check_item !Get_It_Id(it) !Can_Move(it) !Get_Ware(it);
     Item[it1](it)
  []
     it1 !update_item !Get_It_Id(it) ?w: Warehouse_Id ?ip: Item_Place_Id;
     Item[it1](Move(w, ip, it))
  []
     it1 !get_place !Get_It_Id(it) !Get_Place_Id(it);
     Item[it1](it)
  []
     it1 !mark_move !Get_It_Id(it);
     Item[it1](Mark_m(it))
  []
     it1 !mark_remove !Get_It_Id(it);
     Item[it1](Mark_r(it))
  []
     it1 !remove !Get_It_Id(it) !Get_Ware(it) !Get_Place_Id(it);
     stop
endproc (* Item *)

endproc (* Items *)

process Office_Interf[usr3, req2, it1, to_n, from_n](oid: Office_Id) : noexit :=
   ( usr3 !init_remove_item !oid ?it:Item_Id ?t: Time ?wh: Warehouse_Id;
     it1 !check_item !it ?ok:Bool ?w: Warehouse_Id;
     ([not(ok)] -> usr3 !rtn_init_remove_item !oid !ok;
                   exit(oid)
       []
      [ok]  -> ([w ne wh] ->
                   req2 !create_red_req ?id: Red_Req_Id !it !w !wh !t;
                   it1 !mark_move !it;
                   req2 !create_rem_req ?id: Rem_Req_Id !it !w !t;
                   it1 !mark_remove !it;
                   usr3 !rtn_init_remove_item !oid !ok;
                   exit(oid)
                []
                [w eq wh] ->
                   req2 !create_rem_req ?id: Rem_Req_Id !it !w !t;
                   it1 !mark_remove !it;
                   usr3 !rtn_init_remove_item !oid !ok;
                   exit(oid)
               )
     )
   []
     usr3 !init_new_item !oid ?t: Time ?w: Warehouse_Id;
     to_n !reserve_space ?net:Network_Id !w;
     to_n !rtn_reserve_space !net !w ?is_space:Bool;
     ([is_space] ->
          req2 !create_ins_req ?id: Ins_Req_Id !t !w;
          usr3 !rtn_init_new_item !oid !id !true;
          exit(oid)
     []
      [not(is_space)] ->
          usr3 !rtn_init_new_item !oid ?id: Ins_Req_Id !false;
```

```
          exit(oid)
      )
    ) >> accept oid: Office_Id in Office_Interf[usr3, req2, it1, to_n, from_n](oid)
endproc

process Truck_Interf[usr2, tr_r, p_tr](id: Truck_Id) : noexit :=
  (  usr2 !get_trans_req !id;
     p_tr !get_trans_req ?pl: Plan_Id !id;
     exit(id)
   []
     tr_r !request_eta !id ?w: Warehouse_Id ?it:Item_Id;
     usr2 !get_eta !id ?t:Time !w !it;
     tr_r !rtn_request_eta !id !t;
     exit(id)
   []
     tr_r !is_delivered !id ?it: Item_Id ?w: Warehouse_Id;
     usr2 !confirm_delivered !id !it !w;
     tr_r !rtn_is_delivered !id;
     exit(id)
  ) >> accept id: Truck_Id in Truck_Interf[usr2, tr_r, p_tr](id)
endproc (*Truck_Interf*)

process Network[to_n, from_n](id: Network_Id) : noexit :=
   to_n !reserve_space !id ?dest: Warehouse_Id;
   from_n !reserve_space !dest !id;
   from_n !rtn_reserve_space !dest !id ?ok:Bool;
   to_n !rtn_reserve_space !id !dest !ok;
   Network[to_n, from_n](id)
endproc (*Network*)

process Request[req1, req2, p1, p2, tr_r] : noexit :=
    Redis_Requests[req1, req2, p1, p2, tr_r]({} of Red_Req_Id_Set)
    |||
    Insert_Requests[req2]({} of Ins_Req_Id_Set)
    |||
    Remove_Requests[req1, req2, p1, p2]({} of Rem_Req_Id_Set)

where

  process Redis_Requests[req1, req2, p1, p2, tr_r](ids: Red_Req_Id_Set)
     : noexit :=
    req2 !create_red_req ?id: Red_Req_Id ?it: Item_Id
        ?source: Warehouse_Id ?dest: Warehouse_Id
        ?t:Time [(id notin ids)];
    p2 !add_redis_req  ?p: Plan_Id !id;
    (Redis_Request[req1, req2, p1, p2, tr_r]
             (Make_Red_Req(id, it, source, dest, t))
     |||
     Redis_Requests[req1, req2, p1, p2, tr_r](Insert(id, ids))
    )

  where

  process Redis_Request[req1, req2, p1, p2, tr_r](red: Red_Req)
      : noexit :=
     p1 !choose_red !Get_Red_Id(red) ?tr_id: Truck_id;
     tr_r !request_eta !tr_id !Get_From(red) !Get_It_Id(red);
     tr_r !rtn_request_eta !tr_id ?t:Time;
```

```
     req1 !fetch_items !Get_From(red) !Get_It_Id(red) !t;
     tr_r !request_eta !tr_id !Get_Dest(red) !Get_It_Id(red);
     tr_r !rtn_request_eta !tr_id ?t:Time;
     req1 !warn_warehouse !Get_Dest(red) !t;
     tr_r !is_delivered !tr_id !Get_It_Id(red) !Get_Dest(red);
     tr_r !rtn_is_delivered !tr_id;
     req1 !confirm_inserted !Get_Dest(red) !Get_It_Id(red);
     req1 !rtn_confirm_inserted !Get_Dest(red);
     stop
  endproc (*Redis_Request*)

  endproc (*Redis_Requests*)

  process Insert_Requests[req2](ids: Ins_Req_Id_Set)
     : noexit :=
    req2 !create_ins_req ?ins_qid: Ins_Req_Id
         ?t:Time ?w:Warehouse_Id [(ins_qid notin ids)];
    (Insert_Request[req2](Make_Ins_Req(ins_qid, t, w))
     |||
     Insert_Requests[req2](Insert(ins_qid, ids))
    )
  []
    req2 !check_expected ?ins_qid:Ins_Req_Id !false ?w: Warehouse_Id
           ?t: Time [(ins_qid notin ids)];
    Insert_Requests[req2](ids)

  where

  process Insert_Request[req2](ins: Ins_Req) : noexit :=
     req2 !check_expected !Get_Ins_Id(ins) !true !Get_Ware_Id(ins) !Get_Time(ins);
     stop
  endproc (*Insert_Request*)

  endproc (*Insert_Requests*)

  process Remove_Requests[req1, req2, p1, p2](ids: Rem_Req_Id_Set)
     : noexit :=
    req2 !create_rem_req ?id: Rem_Req_Id ?it:Item_Id ?w: Warehouse_Id
            ?t: Time [(id notin ids)];
    p2 !add_rem_req  ?p:Plan_Id !id;
    (Remove_Request[req1, req2, p1, p2](Make_Rem_Req(id, it, w, t))
     |||
     Remove_Requests[req1, req2, p1, p2](Insert(id, ids))
    )

  where

  process Remove_Request[req1, req2, p1, p2](rm: Rem_Req)
     : noexit :=
       p1 !choose_rem !Get_Rem_Id(rm);
       req1 !rm_fetch_items !Get_Ware(rm) !Get_It_Id(rm) !Get_Time(rm);
       stop
  endproc (*Remove_Request*)

  endproc (*Redis_Requests*)

endproc (*Request*)
```

```
   process Planning[p1, p2, p_tr](p: Plan_Id, red_ids: Red_Req_Id_Set,
                       rem_ids: Rem_Req_Id_Set) : noexit :=
         [red_ids ne {}] ->
            p_tr !get_trans_req !p ?tr_id: Truck_Id;
            p1 !choose_red ?red: Red_Req_Id !tr_id;
            Planning[p1, p2, p_tr](p, Remove(red, red_ids), rem_ids)
      []
         [rem_ids ne {}] ->
            (hide removal_time in removal_time;
             p1  !choose_rem ?rem: Rem_Req_Id;
             Planning[p1, p2, p_tr](p, red_ids, Remove(rem, rem_ids))
            )
      []
         p2 !add_rem_req !p ?rem: Rem_Req_Id;
         Planning[p1, p2, p_tr](p, red_ids, Insert(rem, rem_ids))
      []
         p2 !add_redis_req !p ?red: Red_Req_Id;
         Planning[p1, p2, p_tr](p, Insert(red, red_ids), rem_ids)
   endproc


   process Interface_Scenario[usr1, usr2, usr3] : noexit :=
    (* Initialize the system *)
     usr3 !init_new_item ?oid:Office_Id !This_Time ?w: Warehouse_Id;
     usr3 !rtn_init_new_item !oid ?ins_req_id: Ins_Req_Id !true;
     usr1 !insert_query !w !ins_req_id !This_Time;
     usr1 !rtn_insert_query !w !true !ins_req_id;
     usr1 !give_place_item !w ?it_id: Item_Id ?ip: Item_Place_Id;
     usr1 !init_move_item !w !it_id !This_Time ?w2: Warehouse_Id [w ne w2];
     usr1 !rtn_init_move_item !w !true;
     usr2 !get_trans_req ?tr_id: Truck_Id;
     usr2 !get_eta !tr_id !This_Time ?wh: Warehouse_Id ?it1_id: Item_Id;
     usr1 !fetch !wh ?it2_id: Item_Id ?pl_id: Item_Place_Id;
     usr2 !get_eta !tr_id !This_Time ?wh2: Warehouse_Id !it1_id;
     usr1 !warn_arrival !w2 ?t: Time;
     usr2 !confirm_delivered !tr_id !it1_id !wh2;
     usr1 !give_place_item !w2 ?it2_id: Item_Id ?ip2: Item_Place_Id;
     usr3 !init_remove_item !oid !it_id !This_Time !w2;
     usr3 !rtn_init_remove_item !oid !true;
     usr1 !fetch !w2 ?it3_id: Item_Id ?pl2_id: Item_Place_Id;
     (hide success in success; stop)
   endproc (* Interface_Scenario *)

endspec
(*----------------------------------------------------------------------
```

# Appendix C

# LOTOS specification for the car rental system

```
specification My_Rental_System : noexit
  library
    NaturalNumber, Boolean, Set
  endlib

  type Id_Type is Boolean, NaturalNumber
   sorts Id
   opns id1, id2, id3 : -> Id
        h    : Id -> Nat
        _eq_, _ne_, _lt_ : Id, Id -> Bool
   eqns forall n1, n2: Id
     ofsort Nat
       h(id1) = 0;
       h(id2) = succ(h(id1));
       h(id3) = succ(h(id2));
     ofsort Bool
       n1 eq n2 = h(n1) eq h(n2);
       n1 ne n2 = h(n1) ne h(n2);
       n1 lt n2 = h(n1) lt h(n2);
  endtype

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

  type Desk_Id_Set_Type is Set_Id_Type
     renamedby
     sortnames Desk_Id     for Id
               Desk_Id_Set for Set
  endtype

  type Initialization_Id_Set_Type is Set_Id_Type
     renamedby
     sortnames Initialization_Id     for Id
               Initialization_Id_Set for Set
  endtype

  type Reservation_Id_Set_Type is Set_Id_Type
```

```
     renamedby
     sortnames Reservation_Id     for Id
               Reservation_Id_Set for Set
  endtype

  type Contract_Id_Set_Type is Set_Id_Type
     renamedby
     sortnames Contract_Id     for Id
               Contract_Id_Set for Set
  endtype

  type Account_Id_Set_Type is Set_Id_Type
     renamedby
     sortnames Account_Id     for Id
               Account_Id_Set for Set
  endtype

  type Payment_Id_Set_Type is Set_Id_Type
     renamedby
     sortnames Payment_Id     for Id
               Payment_Id_Set for Set
  endtype

  type Client_Id_Set_Type is Set_Id_Type
     renamedby
     sortnames Client_Id     for Id
               Client_Id_Set for Set
  endtype

  type Car_Id_Set_Type is Set_Id_Type
     renamedby
     sortnames Car_Id     for Id
               Car_Id_Set for Set
  endtype

  type Price_Class_Id_Set_Type is Set_Id_Type
     renamedby
     sortnames Price_Class_Id     for Id
               Price_Class_Id_Set for Set
  endtype

  type Branch_Id_Set_Type is Set_Id_Type
     renamedby
     sortnames Branch_Id     for Id
               Branch_Id_Set for Set
  endtype

type Availability_Type is Boolean, NaturalNumber
   sorts Availability
   opns free, reserved, contracted : -> Availability
        h   : Availability -> Nat
        _eq_, _ne_, _lt_ : Availability, Availability -> Bool
   eqns forall n1, n2: Availability
     ofsort Nat
       h(free) = 0;
       h(reserved) = succ(h(free));
       h(contracted) = succ(h(reserved));
     ofsort Bool
```

```
       n1 eq n2 = h(n1) eq h(n2);
       n1 ne n2 = h(n1) ne h(n2);
       n1 lt n2 = h(n1) lt h(n2);
endtype

  type Mileage_Type is
     sorts Mileage
     opns Some_Mileage : -> Mileage
  endtype

  type Car_Type is Car_Id_Set_Type, Availability_Type, Mileage_Type
    sorts State_Car
    opns
      Make_Car     : Car_Id,  Availability, Mileage -> State_Car
      Change_State : State_Car, Availability        -> State_Car
      Get_Mileage  : State_Car -> Mileage
      Get_Car_Id   : State_Car -> Car_Id
  eqns forall c: State_Car, n: Car_Id, v: Availability, m: Mileage
    ofsort Car_Id
      Get_Car_Id(Make_Car(n,v,m)) = n;
      Get_Car_Id(Change_State(c,v)) = Get_Car_Id(c);
    ofsort Mileage
      Get_Mileage(c) = Some_Mileage;
  endtype

  type State_R_Type is Boolean, NaturalNumber
     sorts State_R
     opns ongoing, cancelled, paid, complete : -> State_R
  endtype

  type Money_Type is
     sorts Money
     opns Zero, Some_Amount : -> Money
  endtype

  type Reservation_Type is Reservation_Id_Set_Type, State_R_Type,
                          Money_Type, Boolean
    sorts State_Reservation
    opns
      Make_Reservation : Reservation_Id, State_R, Money -> State_Reservation
      Change_State     : State_Reservation, State_R      -> State_Reservation
      Pay_Deposit      : State_Reservation, Money -> State_Reservation
      Get_Amount       : State_Reservation   -> Money
      Exits_Reservation  : State_Reservation -> Bool
      Get_Reservation_Id : State_Reservation -> Reservation_Id
  eqns forall r: State_Reservation, n: Reservation_Id, s: State_R, a: Money
    ofsort Reservation_Id
      Get_Reservation_Id(Make_Reservation(n,s,a)) = n;
      Get_Reservation_Id(Change_State(r,s)) = Get_Reservation_Id(r);
      Get_Reservation_Id(Pay_Deposit(r,a)) = Get_Reservation_Id(r);
    ofsort Bool
      Exits_Reservation(r) = True;
    ofsort Money
      Get_Amount(r) = Some_Amount;
  endtype

  type State_C_Type is Boolean, NaturalNumber
     sorts State_C
```

```
    opns ongoing, complete : -> State_C
  endtype

  type Contract_Type is Contract_Id_Set_Type, State_C_Type
    sorts State_Contract
    opns
      Make_Contract : Contract_Id, State_C -> State_Contract
      Change_State     : State_Contract, State_C      -> State_Contract
      Get_Contract_Id : State_Contract -> Contract_Id
  eqns forall c: State_Contract, n: Contract_Id, s: State_C
    ofsort Contract_Id
      Get_Contract_Id(Make_Contract(n,s)) = n;
      Get_Contract_Id(Change_State(c,s)) = Get_Contract_Id(c);
  endtype

  type Account_Type is Account_Id_Set_Type, Money_Type
    sorts State_Account
    opns
      Make_Account   : Account_Id, Money   -> State_Account
      Add             : State_Account, Money -> State_Account
      Get_Amount     : State_Account         -> Money
      Get_Account_Id : State_Account         -> Account_Id
  eqns forall a: State_Account, n: Account_Id, m: Money
    ofsort Account_Id
      Get_Account_Id(Make_Account(n,m)) = n;
      Get_Account_Id(Add(a,m)) = Get_Account_Id(a);
    ofsort Money
      Get_Amount(a) = Some_Amount;
  endtype

  type Payment_Type is Payment_Id_Set_Type, Money_Type
    sorts State_Payment
    opns
      Make_Payment   : Payment_Id, Money    -> State_Payment
      Add             : State_Payment, Money -> State_Payment
      Get_Amount     : State_Payment        -> Money
      Get_Payment_Id : State_Payment        -> Payment_Id
  eqns forall a: State_Payment, n: Payment_Id, m: Money
    ofsort Payment_Id
      Get_Payment_Id(Make_Payment(n,m)) = n;
      Get_Payment_Id(Add(a,m)) = Get_Payment_Id(a);
    ofsort Money
      Get_Amount(a) = Some_Amount;
  endtype

  type Op_Names is
    sorts Op_Name
    opns insert, add, reserve, remove, contract, completed, add_car,
         make_reservation, rtn_make_reservation, cancel_reservation,
         create, rent_car, receive_payment, pay_deposit, cancel,
         return_car, extra_payment, add_client, get_mileage, set_free,
         get_amount_paid, ask_rest_payment, reservation_complete,
         rtn_receive_payment, reservation_cancelled, get_reservation,
         rtn_pay_deposit : -> Op_Name
  endtype

behaviour
(*  ( hide d, n, c, c1, c2, r in
```

```
    ( hide a, p, cl, in
      (
       ( Desk [d, cl, a, p, r, c]
        |[cl, a, p]|
         (Account[a] ||| Payment[p] ||| Costumer[cl])
       )
        |[r, c]|
         (Reservation[r, c2] ||| Contract[c, c2])
      )
     |[c2]|
       Car[c1,c2]
    )
   |[c1|]
    ( hide b, pc in
      ( Initialization[n, b, pc, c1]
       |[b, pc]|
        (Branch[b] ||| Price_Class[pc])
      )
    )
  )
  |[d, n]|
    Interface_Scenario[d, n]
*)

   hide d, c, c1, c2, r in
    (( Desk [d, r, c](id1 of Desk_Id)
        |[r, c]|
         ( Reservations[r, c2]({} of Reservation_Id_Set)
         |||
           Contracts[c, c2]({} of Contract_Id_Set)
         )
      )
     |[c1, c2]|
       Cars[c1, c2]({} of Car_Id_Set)
    )
  |[d]|
    Interface_Scenario[d]

  where

  process Desk[d, r, c](id: Desk_Id): noexit :=
    ( d !make_reservation !id ?c_id: Car_Id;
      r !create ?r_id: Reservation_Id !c_id;
      d !rtn_make_reservation !id !c_id !r_id;
      exit (id)
    []
      d !pay_deposit !id ?r_id: Reservation_Id ?m: Money;
      r !pay_deposit !r_id !m;
      d !rtn_pay_deposit !id !r_id;
      exit(id)
    []
      d !cancel_reservation !id ?r_id: Reservation_Id;
      r !cancel !r_id;
      exit(id)
    []
      d !rent_car !id ?r_id: Reservation_Id !id1 of Car_Id;
      r !get_reservation !r_id ?if_res: Bool;
      (*( choice if_res: Bool [] *)
```

```
      (
       [if_res] -> r !reservation_complete !r_id;
                   c !create ?ct: Contract_Id;
                   r !get_amount_paid !r_id ?m: Money;
                              (* add new client ?? *)
                   d !ask_rest_payment !id !m;
                   exit(id)

      []
        [not(if_res)] -> r !cancel !r_id;
                        exit(id)
      )

    []
      d !receive_payment !id ?m: Money;
      d !rtn_receive_payment !id !m;
      exit(id)
 (*   []
      d !return_car
      exit(id)
    []
      d !extra_payment
      exit(id)
    []
      d !add_client
      exit(id)
*)
    ) >> accept id: Desk_Id in Desk[d, r, c](id)
  endproc (* Desk *)

  process Reservations[r, c2](resvs: Reservation_Id_Set): noexit :=
    ( r !create ?r_id: Reservation_Id ?c_id: Car_Id [r_id notin resvs];
      ( Reservation[r, c2]
                 (Make_Reservation(r_id, ongoing of State_R, Zero of Money))
      |||
        Reservations[r, c2](Insert(r_id, resvs))
      )
    )
    where

    process Reservation[r, c2](this_res: State_Reservation): noexit :=
      ( r !pay_deposit !Get_Reservation_Id(this_res) ?m: Money;
        exit(Change_State(Pay_Deposit(this_res,m), paid))
       []
        r !cancel !Get_Reservation_Id(this_res);
        exit(Change_State(this_res,cancelled))
      []
        r !get_reservation !Get_Reservation_Id(this_res)
          !Exits_Reservation(this_res);
        exit(this_res)
      []
        r !get_amount_paid !Get_Reservation_Id(this_res) !Get_Amount(this_res);
        exit(this_res)
      []
        r !reservation_complete !Get_Reservation_Id(this_res);
        exit(Change_State(this_res, complete of State_R))
      ) >> accept this_res: State_Reservation in Reservation[r, c2](this_res)
    endproc (* Reservation *)
```

```
 endproc (* Reservations *)

 process Contracts[c, c2](conts: Contract_Id_Set): noexit :=
   ( c !create ?c_id: Contract_Id [c_id notin conts];
     ( Contract[c, c2](Make_Contract(c_id, ongoing of State_C))
     |||
       Contracts[c, c2](Insert(c_id, conts))
     )
   )
   where

   process Contract[c, c2](this_con: State_Contract): noexit :=
     ( c !completed !Get_Contract_Id(this_con);
       exit(Change_State(this_con, complete of State_C))
     ) >> accept this_con: State_Contract in Contract[c, c2](this_con)
   endproc (* Contract *)
 endproc (* Contracts *)

 process Cars[c1, c2](cs: Car_Id_Set): noexit :=
   ( c1 !create ?c_id: Car_Id [c_id notin cs];
     ( Car[c2](Make_Car(c_id, free of Availability, Some_Mileage))
     |||
       Cars[c1, c2](Insert(c_id, cs))
     )
   )
   where

  process Car[c2](this_car: State_Car): noexit :=
    ( c2 !reserve !Get_Car_Id(this_car);
      exit(Change_State(this_car, reserved of Availability))
    []
      c2 !set_free !Get_Car_Id(this_car);
      exit(Change_State(this_car, free of Availability))
    []
      c2 !contract !Get_Car_Id(this_car);
      exit(Change_State(this_car, contracted of Availability))
    []
      c2 !get_mileage !Get_Car_Id(this_car) !Get_Mileage(this_car);
      exit(this_car)
    ) >> accept this_car: State_Car in Car[c2](this_car)
   endproc (* Car *)
 endproc (* Cars *)

process Interface_Scenario[d] : noexit :=
    d !make_reservation !id1 of Desk_Id !id1 of Car_Id;
    d !rtn_make_reservation !id1 of Desk_Id !id1 of Car_Id
      ?r_id: Reservation_Id;

    d !pay_deposit !id1 of Desk_Id !r_id !Some_Amount of Money;
    d !rtn_pay_deposit !id1 of Desk_Id !r_id;

    d !rent_car !id1 of Desk_Id !r_id !id1 of Car_Id;
    d !ask_rest_payment !id1 of Desk_Id ?m: Money;

    d !receive_payment !id1 of Desk_Id !m;
    d !rtn_receive_payment !id1 of Desk_Id !m;
  (hide success in success; stop)
endproc (* Interface Scenario *)
```

```
endspec
(*---------------------------------------------------------------------
```