

Chapter 7

Formal Object Oriented Development: A Case Study

This chapter reports on a case study which investigated the practical application of FOOD. The goal of the case study was to model the requirements of a simple banking network and, using FOOD, to produce an Eiffel implementation of these requirements. The structure of this chapter is as follows:

- **Section 7.1: Introducing the Banking Network Problem**

This section introduces the case study. The criteria by which the case study was chosen are given. Then, the limitations of the case study are reported. The section concludes by giving an informal overview of the banking network problem, which is the starting point for the development of a formal requirements model.

- **Section 7.2: Formal Object Oriented Analysis of the System**

This section reviews the process by which a formal OO ACT ONE model of the banking network system is developed. In particular, it illustrates how the analysis and synthesis of a formal model improves mutual understanding between customer and analyst. The opportunistic flavour of the analysis and requirements capture method is emphasised.

- **Section 7.3: Design: Moving the System from the Abstract to the Concrete**

This section reviews the process by which the banking network requirements were transformed into a high-level LOTOS specification which was ready for implementation in Eiffel. Particular attention is given to the means by which the internal routing of messages was designed for implementation.

- **Section 7.4: The Eiffel Implementation**

This section records how the Eiffel implementation was developed from the final LOTOS design of the banking network. We emphasise how the analysis and design stages make the implementation process straightforward.

- **Section 7.5: A Review of the Case Study**

This section examines the lessons which arose out of the banking network case study. Three main aspects of the development process are highlighted: the extendibility of systems produced

using FOOD, the production of re-usable components at all stages in FOOD and the need to re-evaluate software development planning due to FOOD placing greater emphasis on the early stage of development.

7.1 Introducing the Banking Network Problem

7.1.1 Choosing the Case Study

A banking network was chosen to form the basis of the case study for the following reasons:

- **Familiarity and Understandability**

The problem of communicating across a network is well understood and there is a wide range of documentation available, for example [109, 102, 39]. Further, the facilities offered by a bank provide a functionality which is accessible to a wide readership.

- **Size and Complexity**

It is important to choose a case study which is large enough to illustrate the FOOD method, whilst small enough to be effectively presented in this thesis. Although the networking concept is very simple, it does deal with complex issues. Further, banking functionality is neither trivial nor overly complex.

- **Extendibility**

There has to be scope within the case study for extending and refining the system requirements. The banking network provides two orthogonal dimensions of complexity which can be extended: the architectural and communicational aspects, and the accounting behaviour.

- **Multidimensional**

It is important that the case study places demands on all three dimensions of software complexity: data structure, data transformation and data communication. Networking is primarily concerned with the communication of data. The banking functionality is complex with respect to the structure and transformation of data. The banking network problem domain provides a case study which naturally combines the complexities of networking and accounting, and so fulfils the multidimensional requirement.

7.1.2 Limitations of the Case Study

The major limitation of the case study is that it does not adequately address the informal aspects of FOOD: in particular, the processes of customer-analyst communication and designer-implementer interaction were not studied. In the case study, the roles of customer, analyst, designer and implementer were played by one person, namely the author¹. The case study provides a good evaluation

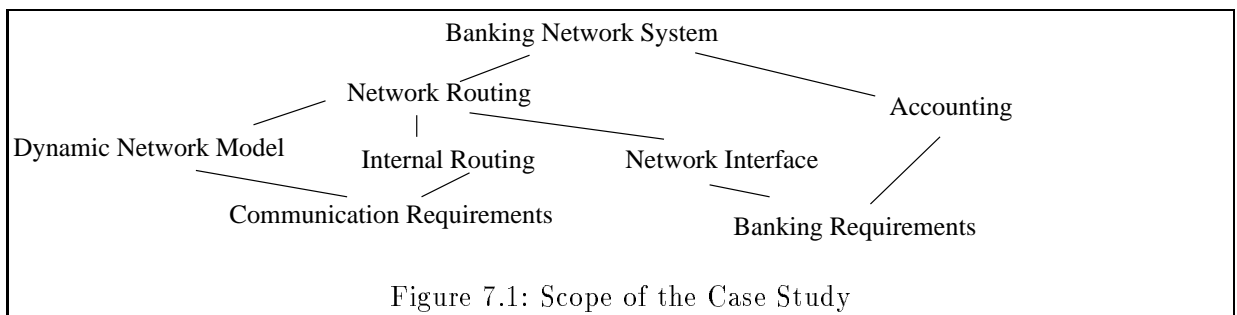
¹In a preliminary investigation, a network routing model was developed in a project involving both a specifier (the author) and an implementer (David Freer, of British Telecom, must be thanked for his contribution in the development of the C++ code). [55] reviews this preliminary investigation and discusses the process of specifier-implementer communication. It concludes by stating that this communication is improved by the synthesis and analysis of a formal model of requirements.

of the mathematical models, but only a limited evaluation of the methods in FOOD. We believe that a method should evolve from repeated use of models rather than sprouting automatically from the theory. The application of FOOD in one case study does not justify the definition of a prescriptive development method.

The size of the case study was also a limitation: it was not possible to investigate all aspects of the FOOD approach. The case study does, however, illustrate the application of FOOD in a non-trivial problem domain. There is good reason to believe that, given the object oriented nature of FOOD, it can be applied to even larger systems, requiring the co-ordinated attention of groups of developers.

7.1.3 The Scope of the Problem: An Informal Overview of Requirements

The scope of the requirements is represented in figure 7.1. An informal description of the problem domain is given in sections 7.1.3.1 to 7.1.3.4. This informal description forms the basis upon which the analysis and synthesis of the requirements model are initiated.



7.1.3.1 Dynamic Network Model

It is a requirement of the system that the network is composed of a set of nodes and a set of links, each of which must connect two of the nodes. The topology of the network must be dynamic: it must be possible to add nodes and links during system execution. The links provide communication lines between nodes along which banking transactions are routed.

7.1.3.2 Network Interface

The interface to the network system is constructed from the interfaces of each of the node components. There are four different types of node, each of which offers its own type of interface:

- **Relay Node**

A relay node acts purely as a communication buffer for the receiving, routing and sending of messages. Relay nodes offer an interface to system engineers to permit their switching on and off: nodes which are off cannot service requests. All nodes in the system are uniquely identified as relay nodes.

- **Automatic Teller Machine (ATM)**

An ATM provides an interface to account holders for the reading and writing of account details in response to deposit, withdrawal and enquiry transactions.

- **Control Node**

A control node provides an interface to bank employees for the purpose of account maintenance. They also act as the access points to a subset of the database of account information. Every account transaction must be routed to one of the control nodes in the network for processing.

- **Teller Node**

A teller node offers all the functionality of a control node and an ATM. These are used when account holders and bank employees access account information together.

7.1.3.3 Internal Routing of Accounting Information

All account transactions must be routed through a specified control node. We do not require that every transaction is processed, but an *internal* timeout facility must inform the customer (account holder or bank employee) when a transaction has not been processed. The means by which transactions are routed to/from control nodes is a design and implementation decision.

7.1.3.4 Accounting

The database of accounts can be altered by the control nodes in the following ways:

- **Creating a new account**

A new account is created when the appropriate details are provided, depending on the account type (see below).

- **Closing an existing account**

An account, specified by a given identification, can be closed only when the balance is zero.

- **Changing the restrictions on an account**

A **restricted** account (see below) has a limit placed on the size of individual withdrawals. This limit can be changed at a control node.

- **Changing the overdraft facility**

The overdraft limit for any given account be changed at a control node.

There are three types of account:

- A **basic** account permits the customer to deposit and withdraw money. Further, the customer can request details of the amount of money available to them (i.e. their balance plus their overdraft limit).
- A **business** account permits the customer to deposit and withdraw money, and request a statement of their last three money transactions or the current amount available.
- A **restricted** account permits the customer to deposit and withdraw money. The customer is restricted to withdrawing no more than a predefined amount at any one transaction. A statement of the amount available can also be given for a **restricted** account.

Customers interact with their accounts, which must be uniquely identifiable, from the ATM nodes of the banking network.

7.2 Formal Object Oriented Analysis of the System

The informal requirements, given in the previous section, provide a good overview of the scope of the problem. They also provide the basis upon which a formal requirements model can be constructed. The construction of the formal requirements model plays three main roles:

- It improves customer and analyst understanding of the problem.
- It provides an executable model for customer validation.
- It acts, in its final form, as an input to the design stage.

7.2.1 *What not How*

The formal object oriented analysis language, OO ACT ONE, is used to define the accounting functionality offered at the external interface of the banking network system. Further, OO ACT ONE is used to model the dynamic network requirements: *how* these are realised is not an analysis concern. OO ACT ONE is not used to specify *how* the internal routing of transactions, from ATM node to control node, takes place.

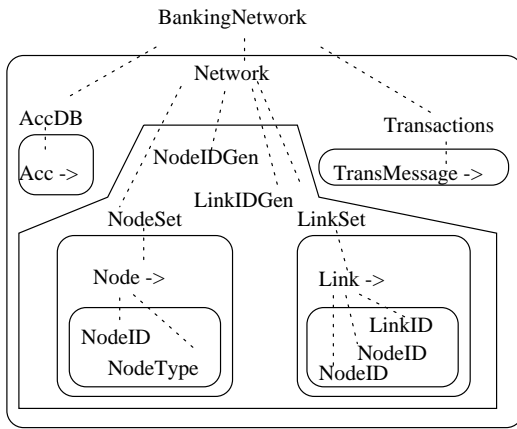
7.2.2 Applying the Skeleton Method to Requirements Capture

Section 4.5.2 defines a *skeleton* method for the synthesis and analysis of requirements models in OO ACT ONE. An important aspect of this method is customer interaction. Although there was no direct customer involvement in the case study, the process of *customer interaction* was replaced by the need for the requirements model to be tested against relevant documentation and intuition. Consequently, in the remainder of this chapter, when referring to *customer interaction* it is this testing process to which we are alluding.

Other than *customer interaction*, all other parts of the *skeleton* method were carried out as intended. The opportunistic aspect of the method meant that there were many ways in which the requirements model could have been constructed and validated. The sequence of steps which was followed is reported below.

Step 1: Composition Analysis of the BankingNetwork System Class

To start, there is only one class to be considered: the **BankingNetwork**. From the informal requirements it is clear that a composition analysis will improve ones understanding of the requirements. Consequently, the analysis is started with a structured decomposition of the problem based on the *has-a* relationship. The diagram below illustrates the initial decomposition of the banking network.



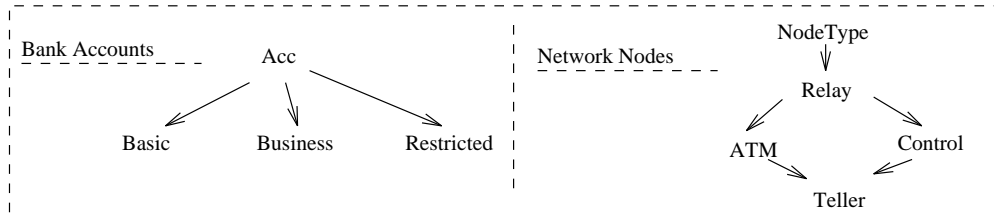
A BankingNetwork is composed from:

- An account database (of class AccDB) which is composed from:
 - Accounts (of class Acc) in a recursive structure
- A network (of class Network) which is composed from:
 - A set of nodes (of class NodeSet) which is composed from:
 - Nodes (of class Node) in a recursive structure, each Node composed from:
 - A unique identifier (of class NodeID)
 - A type identifier (of class NodeType)
 - A generator of unique node identifications (NodeIDGen)
 - A set of links (of class LinkSet) which is composed from:
 - Links (of class Link) in a recursive structure, each Link composed from:
 - A unique identifier (of class LinkID)
 - Two node identifiers (of class NodeID)
 - A generator of unique link identifications (LinkIDGen)
 - A transaction set (of class Transactions) which is composed from:
 - Transactions (of class TransMessage) in a recursive structure

Note: A recursive structure, represented as Element ->, is the means in OO ACT ONE of modelling linked lists of Elements.

Step 2: Classification Analysis

Examination of the informal requirements leads to the initial identification of two² subclass hierarchies: account classes, specified as class `Acc`, and node classes, specified as class `NodeType`. It is important that these two subclassing relationships are explicitly defined at some point in the requirements capture.



Step 3: Analysis and Synthesis of Network Requirements Model

The network component of the system appears, from the composition analysis, to play a major role in providing the behaviour of the banking network. Consequently, we chose to analyse and synthesise a network requirements model. The behaviour of the network component is defined by an OO ACT ONE `Network` class³.

Step 3.1: The External Interface of the Network

The composition of the `Network` class has already been analysed in step 1. It is necessary now to consider how the components of `Network` combine to provide network behaviour. An initial OO ACT ONE specification of the `Network` class facilitates further investigation of the requirements. The `Network` header, below, defines the external interface of the class. The header also defines a `Network`

²Although there are only two hierarchies identified at this stage, this does not mean that there are only two hierarchies in the system. Step 4.4 identifies another hierarchy which was 'missed' in this early stage. The opportunistic approach to building a formal requirements model encourages the analyst to record understanding even when it is incomplete.

³By convention, all class identifiers in the case study have an initial capital letter.

to have a fixed structure⁴, i.e. a static set of component classes.

```

CLASS Network USING NodeSet, LinkSet, NodeIDGen, LinkIDGen OPNS
STRUCTURES: ANetwork<NodeSet,LinkSet,NodeIDGen,LinkIDGen> (*FIXED*)
TRANSFORMERS: addNode<NodeType>, addLink<NodeID,NodeID>, switch<NodeID>
ACCESSORS: getNode<NodeID> -> Node, isNode<NodeID> -> Bool,
areConnected<NodeID,NodeID> -> Bool EQNS ...

```

Step 3.2: The Behaviour at the External Interface

The behaviour intended for each of the **Network** attributes⁵ is as follows:

- **addNode**: Takes a **NodeType** as an input parameter and allocates it a new identifier (provided by the **NodeIDGen** component). These two values are made into a single **Node**, which is then added to the **NodeSet**.
- **switch**: Takes a **NodeID** as an input parameter and switches the corresponding **Node** in the **NodeSet** from **off** to **on** (or vice versa).
- **getNode**: Takes a **NodeID** as an input parameter and returns the corresponding **Node** in the **Network**, and **false** otherwise.
- **isNode**: Takes a **NodeID** as input parameter and returns **true** if there is a **Node** in the **NodeSet** with **NodeID** as its identifier.
- **areConnected**: Takes two **NodeIDs** as input parameters and returns **true** if there is a **Link** in the **LinkSet** which connects the two specified **Nodes**.
- **addLink**: Takes two **NodeIDs** as input parameters and adds a new **Link** to the **LinkSet**. The new **Link** is allocated a unique identifier by the **LinkIDGen** component.

When a **NodeID** input parameter does not identify a node in the **NodeSet** of the network then an exception must be defined. (All exceptions must be dealt with during design and implementation.)

A class structure diagram (see figure 7.2) is used to show how the **Network** depends on each of its components to fulfil its external functionality.

⁴By convention, in the case study, all fixed structures of a class **ClassName** are labelled **AClassName**.

⁵The attribute labels have, by convention, a non-capital initial.

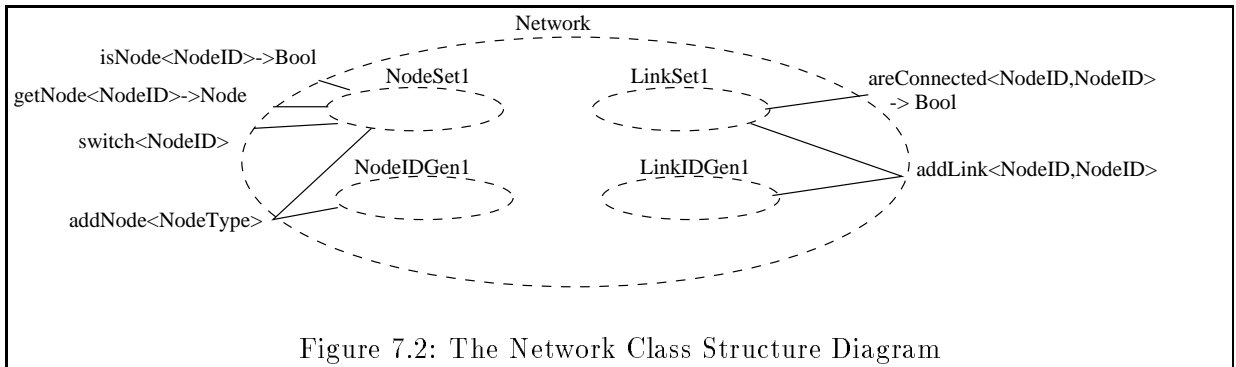


Figure 7.2: The Network Class Structure Diagram

The OO ACT ONE equation definitions for the **Network** class are given below⁶.

```

CLASS Network ... (* Header is given above *) EQNS
ANetwork(...).addNode(NodeType1) =
  ANetwork(NodeSet1.add(NodeIDGen1..nextN,NodeType1),LinkSet1,NodeIDGen1.nextN,LinkIDGen1);
((NodeSet1..isNode(NodeID1)).and(NodeSet1..isNode(NodeID2))).not =>
  ANetwork(...).addLink(NodeID1,NodeID2) = ~NodeSet OTHERWISE
  ANetwork(NodeSet1,LinkSet1.link(NodeID1,NodeID2,LinkIDGen1..nextL),
  NodeIDGen1,LinkIDGen1..nextL);
(NodeSet1..contains(NodeID1)).not =>
ANetwork(...).switch(NodeID1) = ~NodeSet OTHERWISE
ANetwork(NodeSet1.switch(NodeID1), LinkSet1, NodeIDGen1, LinkIDGen1);
(NodeSet1..contains(NodeID1)).not =>
ANetwork(...)..getNode(NodeID1) = ~NodeSet OTHERWISE (NodeSet1.getNode(NodeID1))..getNode;
ANetwork(...)..isNode(NodeID1)= NodeSet1..contains(NodeID1);
ANetwork(...)..areConnected(NodeID1, NodeID2) = LinkSet1..areConnected(NodeID1, NodeID2)
ENDCLASS (* Network *)

```

Step 3.3: Analysis of the IDGen Classes

One class, **IDGen**, is defined for the generation of unique identifiers. It is defined to have a **DUAL** attribute, **next**, for the generation of unique identifiers, and an **ACCESSOR** attribute, **eq**, for testing the equality of identifiers. **NodeIDGen** and **LinkIDGen** are defined as renamings of **IDGen**.

IDGen must be able to generate an infinite number of identifiers. The simplest way of specifying this is to define an ID class with a recursive **STRUCTURE** operation, together with a base **LITERAL** value. Rather than specifying **IDGen** as a store of the previously allocated IDs, a standard scheme is employed whereby a unique identifier can always be generated when only the previously allocated identifier is known. Classes **ID** and **IDGen** are defined below. These classes are added to a library for re-use.

NodeID is defined, below, using the OO ACT ONE renaming construct.

⁶A simple syntactic sugar is used in the remainder of this chapter to simplify the representation of OO ACT ONE equation definitions in a class with a fixed structure. Rather than writing **Structure**(par1,...,parn) = ... on the left hand side of equation definitions, **Structure**(...) = ... is used without risk of ambiguity.

```

CLASS ID USING Bool OPNS
LITERALS: 0 STRUCTURES: IDSt<ID>
ACCESSORS: eq<ID> -> Bool
EQNS 0..eq(0) = true; 0..eq(IDSt(ID1)) = false; IDSt(ID1)..eq(0) = false;
IDSt(ID1)..eq(IDSt(ID2)) = ID1..eq(ID2)
ENDCLASS (* ----- ID ----- *)
CLASS IDGen USING ID EXTENDS ID WITH OPNS
DUALS: Next -> ID
EQNS IDGen1.Next = IDSt(IDGen1) AND IDGen1
ENDCLASS (* IDGen *)

```

```

CLASS NodeID RENAMES ID LITERALS: 0 WITH NO STRUCTURES: IDSt WITH N
ENDCLASS (* ----- NodeID ----- *)
CLASS NodeIDGen USING NodeID EXTENDS NodeID WITH OPNS DUALS: NextN -> NodeID
EQNS NodeIDGen1.NextN = N(NodeIDGen1) AND NodeIDGen1
ENDCLASS (* NodeIDGen *)

```

This simple example illustrates the limited use of the `RENAMES` facility. It is not possible to rename both `ID` and `Gen` to create `NodeID` and `NodeIDGen` because the subsequent subclassing relationship between these two classes will not be properly defined. Consequently, it is necessary to define `NodeID` as a renaming of `ID`, and `NodeIDGen` as an extension of `NodeIDGen`⁷. `LinkIDGen` and `LinkID` are also defined similarly: the `STRUCTURE` operation is renamed `L`, the `LITERAL` is renamed `LO` and the `DUAL` is renamed `NextL`.

Step 3.4: Analysis of the NodeSet Class

`NodeSet` is required, by the `Network`, to offer the following external attributes:

- **TRANSFORMER:** `add< NodeID, NodeType >`
Create a node from identifier and type components, and add it to the `NodeSet`.
- **TRANSFORMER:** `switch< NodeID >`
Search the node set for the node identified by the `NodeID` parameter and switch the state of this node from `on` to `off` (or vice versa).
- **ACCESSOR:** `isNode< NodeID > -> Bool`
Return `true` if there is a node in the network which is identified by `NodeID`.
- **ACCESSOR:** `isOn< NodeID > -> Bool`
Return `true` if there is a node in the network identified by `NodeID` which is `on`, otherwise return `false` if the identified node is `off`.

The `NodeSet` class definition is given below.

⁷An obvious extension to OO ACT ONE is to provide a more powerful renaming facility which ‘copies’ class hierarchies rather than individual classes. The investigation of the semantics of such a copy is beyond the scope of this thesis.

```

CLASS NodeSet USING Node OPNS ... (* Operations as specified above *)
EQNS emptyNodeSet..isNode(NodeID1) = false;
(Node1.getID).eq(NodeID1) =>
NodeStr(Node1, NodeSet1)..isNode(NodeID1) = true OTHERWISE NodeSet1..isNode(NodeID1);
emptyNodeSet..getNode(NodeID1) = ~Bool; emptyNodeSet..switch(NodeID1)= ~NodeSet;
(Node1.getID).eq(NodeID1) => NodeStr(Node1, NodeSet1)..getNode(NodeID1) =
  Node1 OTHERWISE NodeSet1..isOn(NodeID1);
NodeSet1.add(NodeID1, NodeType1) = NodeSetStr(ANode(NodeID1, NodeType1), NodeSet1);
(Node1.getID).eq(NodeID1) => NodeSetStr(Node1, NodeSet1)..switch(NodeID1) =
  NodeSetStr(Node1..switch, NodeSet1) OTHERWISE NodeSet1.isNode(NodeID1) =>
  NodeSetStr(Node1, NodeSet1.switch(NodeID1)) OTHERWISE ~NodeSet
ENDCLASS (* NodeSet *)

```

Step 3.5: Analysis of the Node Class

The composition of the `Node` class has already been identified as a fixed `STRUCTURE` (`ANode`) of two components: a `NodeID` and a `NodeType`. The `NodeSet` class places requirements on the external interface of `Node` which are fulfilled by the OO ACT ONE specification given below.

```

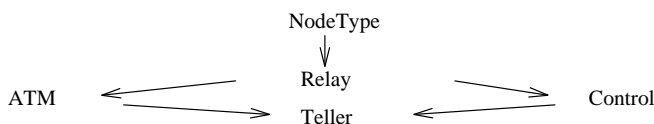
CLASS Node Using NodeID, NodeType
STRUCTURES: ANode<NodeID, NodeType> (*FIXED*)
ACCESSORS: isOn -> Bool, isControl -> Bool, isATM -> Bool, getID -> NodeID
TRANSFORMERS: switch
EQNS ANode(...)..isOn = NodeType1..isOn; ANode(...)..isATM = NodeType1..isATM;
ANode(...)..isControl = NodeType1..isControl; ANode(...)..getID = NodeID1; ANode(...).switch =
ANode(NodeID1, NodeType1.switch)
ENDCLASS (* Node *)

```

The `isOn`, `isControl`, `isATM` and `switch` service requests are ‘passed on’ to the `NodeType` component. The `getID` attribute returns the node identifier. The `NodeID` class has already been specified and so we now consider the `NodeType` class.

Step 3.5.1 Analysis of the NodeType class

The `NodeType` class has been identified as the root of a subclass hierarchy:



At the analysis stage of this case study, the precise role of the `NodeType` classes is not specified. The only requirements placed on `NodeType` objects is that they can be either `on` or `of`, and offer `ACCESSOR` attributes `isOn`, `isControl` and `isATM`. The `NodeType` class (and its subclasses) are specified in OO ACT ONE, below.

```

CLASS NodeType Using Bool OPNS
LITERALS: on, off
ACCESSORS: isOn -> Bool
EQNS on..isOn = true; off..isOn = false ENDCLASS (* NodeType *)
CLASS Relay USING NodeType EXTENDS NodeType WITH OPNS TRANSFORMERS: switch
EQNS on.switch = off; off.switch = on ENDCLASS (* Relay *)
CLASS ATM USING Relay EXTENDS Relay WITH OPNS ACCESSORS: isATM -> Bool
EQNS ATM1..isATM = true ENDCLASS (* ATM *)
CLASS Teller USING ATM EXTENDS ATM WITH OPNS ACCESSORS: isControl -> Bool, isTeller -> Bool
EQNS Teller..isControl = true; Teller..isTeller = true ENDCLASS (* Teller *)
CLASS Control USING Teller RESTRICTS Teller TO OPNS
ACCESSORS: isControl TRANSFORMERS: switch
ENDCLASS (* Control *)

```

Step 3.6 Analysis of LinkSet Class

The `LinkSet` class has been identified as recursive structure of `Links`, with `ACCESSOR` attribute `areConnected` and `TRANSFORMER` attribute `link`. The behaviour of `LinkSet` is formally defined below.

```

CLASS LinkSet Using Link OPNS
LITERALS: emptyLinkSet STRUCTURES: LinkStr<Link, LinkSet>
ACCESSORS: areConnected< NodeID, NodeID > -> Bool
TRANSFORMERS: link < NodeID, NodeID >
EQNS LinkSet1.link(NodeID1, NodeID2, LinkID) = LinkStr(ALink(NodeID1,NodeID2,LinkID), LinkSet1);
empty..areConnected(NodeID1, NodeID2) = false;
(Link1..conn1)..eq(NodeID1)).and((Link1..conn2)..eq(NodeID2)).or(
(Link1..conn2)..eq(NodeID1)).and((Link1..conn1)..eq(NodeID2))) =>
LinkStr(Link1,LinkSet1)..areConnected(NodeID1,NodeID2)= true OTHERWISE
LinkSet1..areConnected(NodeID1,NodeID2)
ENDCLASS (* LinkSet *)

```

The `Link` class is a simple passive holder of data in a fixed structure. It has three external attributes for accessing the values of each of its three components. The OO ACT ONE specification of `Link` is given below.

```

CLASS Link Using LinkID, NodeID OPNS
STRUCTURES: ALink<NodeID, NodeID, LinkID> (*FIXED*)
ACCESSORS: conn1 -> NodeID, conn2 -> NodeID, getID -> LinkID
EQNS ALink(...)..conn1 = NodeID1; ALink(...)..conn2 = NodeID2; ALink(...)..getID = LinkID1
ENDCLASS (* Link *)

```

Step 3.7: Customer Validation of Network

The OO ACT ONE specification of the `Network` class is now put forward for *customer validation*. The executable ACT ONE `Network` model is successfully generated and this is used to test the `Network` requirements model. *Customer validation* of the `Network` resulted in two changes being made to the `Network` requirements model. Firstly, an invariant property was added to the `Link`

class to specify that a link cannot connect a node to itself: `ALink(NodeID1, NodeID2, LinkID1) REQUIRES (NodeID1..eq(NodeID2))..not`. Secondly, in order to ensure the correct addition of links to the network, an exception was defined to occur when a request is made to connect a node to itself: `NodeID1.eq(NodeID2) => Network1.addLink(NodeID1, NodeID2) = ~Network OTHERWISE (* as before *)`.

The process of validation brought the *quality* of the requirements model into question:

- **Question:** Could `NodeSet` and `LinkSet` be *better* defined as instances of a generic `Set`?

Answer: The `NodeSet` and `LinkSet` classes provide quite specific behaviour: different types of access to (and transformation of) individual set elements. A standard generic set definition is not suitable for the parameterisation of these two behaviours.

- **Question:** Could the `Network` behaviour be defined, for re-use, in a generic class?

Answer: Generic network behaviour can be usefully defined: graphs of nodes and links are common in computing systems. This is noted for future investigation and development.

- **Question:** Is the *customer view* of the network as having separate `NodeSet` and `LinkSet` components the *best* way of conceptualising (and communicating) the requirements?

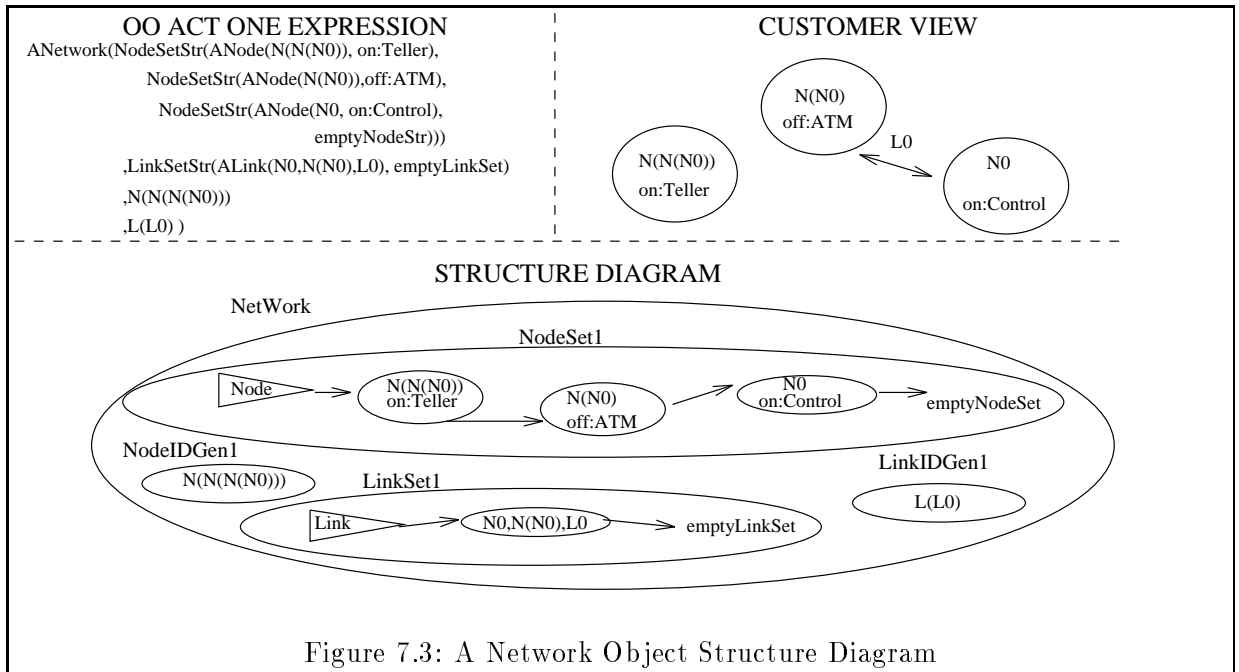
Answer: As the requirements model was developed and validated, a better understanding of the `Network` evolved. It was felt that a better understanding of the requirements could have been achieved if the nodes in the network had been specified as ‘knowing’ the other nodes to which they were connected, rather than having a separate `LinkSet`. This type of reconceptualisation must always be proposed to the *customer*: only after they agree that the new model is a *better* recording of their understanding of the requirements can appropriate changes be made. In the case study, for reasons explained at the beginning of this chapter, the process of *customer validation* could not be properly evaluated, and the process of reconceptualisation was not carried out. Examination of the process of communication between customer and analyst, particularly the influence of the analyst on the way in which the customers conceptualise their requirements, is beyond the scope of this thesis.

Step 3.7: Provide a Graphical View of Network State

The OO ACT ONE requirements model provides an excellent statement of `Network` behaviour, which can be supplemented by graphical views. It is often advantageous (particularly for complex classes of behaviour) to record, in the requirements documentation, the correspondence between:

- The OO ACT ONE representation of a class member.
- The structure diagram of a class member.
- The customer’s conceptualisation of a class member.

When it is clear that the customer’s view of the requirements provides a useful way of representing the behaviour of the system then an attempt should be made to provide a formal semantics, founded on the OO ACT ONE specification, of their representation. An example of this is given in figure 7.3, where the customer’s view of a particular network is given a formal meaning.



Step 3.7: Make The Network Available For Design

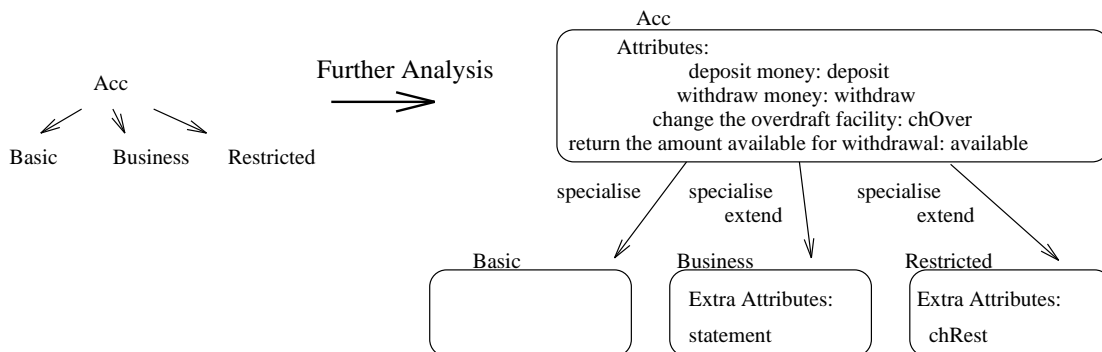
After *customer validation*, the **Network** is ready for design and implementation. The decision to proceed with its development must be taken by the project managers. The risk of developing the **Network** before the analysis of the **BankingNetwork** is complete must be weighed against the advantages of running design and implementation in parallel with the unfinished analysis and requirements process. This risk must be evaluated for every class in the system being analysed.

Step 4: Analysis of the AccDB Class and Synthesis of the Requirements Model

After the **Network**, the account database (**AccDB**) appears to play the next most significant role in the behaviour of the banking network. The **AccDB** class is a recursive structure of **Acc** classes. Before we analyse **AccDB** behaviour we analyse the behaviour of its **Acc** components.

Step 4.1 Analysis of Acc Behaviour

It is useful to analyse the relationship between **Acc** and each of its subclasses, namely **Basic**, **Business** and **Restricted**. Further analysis of the informal requirements leads to a more informative class hierarchy:



Every account has identifier, balance and overdraft components. Restricted accounts have a restriction on the amount of money which can be withdrawn in one transaction. Business accounts can supply, on request, a statement of the last three transactions which the account has processed. The balance, overdraft and restriction components are defined to belong to the class `Sum`, which represents an amount of money (positive or negative).

As a result of this analysis, three new classes are identified:

- `AccID`, which defines a means of uniquely identifying accounts. This is defined as a renaming of `ID`: the LITERAL `0` is renamed `A0`, the STRUCTURE `IDst` is renamed `A` and the DUAL `Next` is renamed `NextA`.
- `Trans3X`, which defines a record of the last three transactions that a business account has serviced. This class of behaviour is synthesised and analysed in step 4.3.
- `Sum` provides a means of recording positive and negative amounts of money. It also provides the necessary arithmetic for the manipulation and testing of these amounts. The class `Sum` can be defined as a renaming of some standard class of *numbers*⁸.

Step 4.1.1 An initial `Acc` model

A sequence of OO ACT ONE `Acc` models were developed to improve understanding of the requirements. The first such model is defined below (as version⁹1).

Analysis of version 1 of the `Acc` requirements model gives rise to a number of questions which must be answered by the *customer*:

- i) What happens when a withdrawal is requested which is greater than the amount available?
- ii) What happens when a withdrawal is requested of a restricted account which exceeds the restriction?
- iii) What happens when a change of the overdraft facility is requested which would result in the overdraft being exceeded?

⁸Like ACT ONE, OO ACT ONE is not well suited to representing numerical behaviour. The definition of *numbers* is not reported in this thesis.

⁹Often it is necessary to construct a prototype model (or models) as a means of improving the mutual understanding between customer and analyst. These prototypes must be clearly distinguished from the final requirements models: it is recommended that every prototype is given a version number and the documentation include details of what was learned from the analysis of each version.

```

CLASS Acc Using AccID, Sum, Trans3X OPNS (* Version 1 *)
STRUCTURES: BasicStr<AccID, Sum (*balance*), Sum (*overdraft*) >,
BusinessStr<AccID, Sum (*balance*), Sum (*overdraft*), Trans3X>,
RestStr<AccID, Sum (*balance*), Sum (*overdraft*), Sum (*restriction*)>
ACCESSORS: available -> Sum, getID -> AccID
TRANSFORMERS: deposit<Sum>, withdraw<Sum>, chOver<Sum>
EQNS BasicStr(AccID1,Sum1,Sum2)..available = Sum1.add(Sum2);
BasicStr(AccID1,Sum1,Sum2)..getID = AccID1;
BasicStr(AccID1,Sum1,Sum2).chOver(Sum3) = BasicStr(AccID1,Sum1,Sum3);
(* available, getID and chOver are defined similarly for the other structures *)
BasicStr(AccID1,Sum1,Sum2).deposit(Sum3) = BasicStr(AccID1,Sum1.add(Sum3),Sum2);
BasicStr(AccID1,Sum1,Sum2).withdraw(Sum3) = BasicStr(AccID1,Sum1.sub(Sum3),Sum2);
BusinessStr(AccID1,Sum1,Sum2,Trans3X1).deposit(Sum3) =
    BusinessStr(AccID1,Sum1.add(Sum3),Sum2,Trans3X1.insert(ATrans(DepositStr(Sum3))));
BusinessStr(AccID1,Sum1,Sum2,Trans3X1).withdraw(Sum3) =
    BusinessStr(AccID1,Sum1.sub(Sum3),Sum2,Trans3X1.insert(ATrans(WithdrawStr(Sum3))));
RestStr(AccID1,Sum1,Sum2,Sum3).deposit(Sum4) = RestStr(AccID1,Sum1.add(Sum4),Sum2,Sum3);
RestStr(AccID1,Sum1,Sum2,Sum3).withdraw(Sum4) = RestStr(AccID1,Sum1.sub(Sum4),Sum2,Sum3);
ENDCLASS (* Acc *)

```

- iv) Can the overdraft, restriction, deposits and withdrawals be negative? If so, what happens in each case?

Step 4.1.2: Backtracking

The analysis also identified a misrepresentation in the way in which the `available` attribute is defined for restricted accounts: the restriction amount should be returned when this is smaller than the sum of the balance and overdraft amounts. This misrepresentation was corrected by making a change in the final version of the `Acc` requirements model. The `Acc` model was also changed to record the following additional requirements:

- All the scenarios in questions (i) to (iii) result in exceptions which must be dealt with by designers and/or implementers.
- A negative overdraft is used to model a minimum amount that must be kept in an account.
- A negative restriction is not permitted by the definition of an invariant property, and a request to change a restriction to a negative amount results in an exception.
- Requests to deposit or withdraw negative amounts are also defined as exception cases.
- Additional `ACCESSOR` attributes are defined to test the type of a given account.
- An additional `ACCESSOR zeroBalance` is defined to return true if the balance of an account is zero. This test is needed for closing accounts.

Step 4.1.3 A Final Acc model

The new version of the `Acc` requirements is re-tested and *customer validated*. This final version of the model is defined below.

```

CLASS Acc Using AccID,TransResult,Sum,Trans3X (*FINAL*) STRUCTURES: (*As in version 1*)
ACCESSORS: available -> Sum, getID -> AccID,
zeroBalance -> Bool, isBasic -> Bool, isBusiness -> Bool, isRest -> Bool;
INVARIANTS: Reststr(AccID1,Sum1,Sum2,Sum3) REQUIRES Sum3..positive
EQNS (* available, getID and chOver: as defined in version 1 *)
BasicStr(AccID1, Sum1, Sum2)..isBasic = true; (* other structures defined similarly *)
BasicStr(AccID1, Sum1, Sum2)..isBusiness = false; (* other structures defined similarly *)
BasicStr(AccID1, Sum1, Sum2)..isRest= false; (* other structures defined similarly *)
BasicStr(AccID1, Sum1, Sum2)..zeroBalance= Sum1.eq(0); (* other structures defined similarly *)
Sum3..positive => BasicStr(AccID1, Sum1, Sum2).deposit(Sum3) =
  BasicStr(AccID1, Sum1.add(Sum3), Sum2) OTHERWISE ~Acc;
Sum3..positive => BasicStr(AccID1, Sum1, Sum2).withdraw(Sum3) =
  BasicStr(AccID1, Sum1.sub(Sum3), Sum2) OTHERWISE ~Acc;
Sum3..positive => BusinessStr(AccID1, Sum1, Sum2, Trans3X1).deposit(Sum3) =
  BusinessStr(AccID1,Sum1.add(Sum3),Sum2,Trans3X1.insert(ATrans(DepositStr(Sum3))))
  OTHERWISE ~Acc Sum3..positive => BusinessStr(AccID1,Sum1,Sum2,Trans3X1).withdraw(Sum3) =
  BusinessStr(AccID1,Sum1.sub(Sum3),Sum2,Trans3X1.insert(ATrans(WithdrawStr(Sum3))))
  OTHERWISE ~Acc;
Sum4..positive => RestStr(AccID1,Sum1,Sum2,Sum3).deposit(Sum4) =
  RestStr(AccID1,Sum1.add(Sum4),Sum2,Sum3) OTHERWISE ~Acc;
Sum4..positive => RestStr(AccID1,Sum1,Sum2,Sum3).withdraw(Sum4) =
  RestStr(AccID1,Sum1.sub(Sum4),Sum2,Sum3) OTHERWISE ~Acc ENDClass (* Acc *)

```

Step 4.2 Analysis and Synthesis of Acc Subclasses

The **Basic** class is defined as a specialisation of **Acc**, restricted to values represented by the **BasicStr** STRUCTURE. The **Business** and **Restricted** classes are defined to specialise and extend **Acc**. These classes are defined below. Their behaviours, being based on the already validated **Acc** class, are easy to check with the customer. An important point to note is that **Business** and **Restricted** are not defined as subclasses of **Basic**. These classes could have been defined in this way but it was clear that the three different types of account were intended to be unrelated: for example, a business account is not a basic account.

```

CLASS Basic USING Acc SPECIALISES ACC TO OPNS STRUCTURES: BasicStr
ENDCLASS (* Basic *)
CLASS Business USING Acc SPECIALISES AND EXTENDS ACC TO OPNS
STRUCTURES: BusinessStr ACCESSORS: statement -> Trans3X
EQNS BusinessStr(AccID1, Sum1, Sum2, Trans3X1)..statement= Trans3X1
ENDCLASS (* Business *)
CLASS Restricted USING Acc SPECIALISES AND EXTENDS ACC TO OPNS
STRUCTURES: RestStr TRANSFORMERS: chRest<Sum>
EQNS Sum4..positive => RestStr(AccID1, Sum1, Sum2, Sum3).chRest(Sum4) =
  RestStr(AccID1, Sum1, Sum2, Sum4) OTHERWISE ~RestStr
ENDCLASS (* Restricted *)

```

Step 4.3: Analysis of Trans3X Class

This class is required to record the last three transactions which have been serviced on a business account. A transaction is added using the `insert` attribute. The most recent transaction is stored as the first component, and the least recent as the third component, of a fixed `STRUCTURE`. This behaviour is specified below.

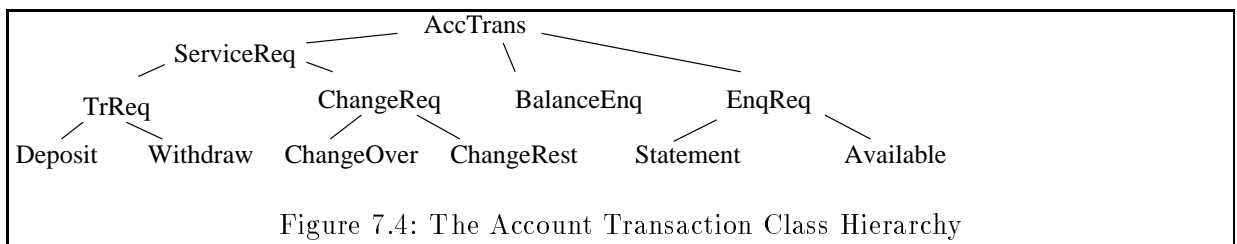
```

CLASS Trans3X USING TrReq OPNS
STRUCTURES: ATrans3X< TrReq, TrReq, TrReq > (*FIXED*)
TRANSFORMERS: insert<TrReq>
EQNS ATrans3X(TrReq1, TrReq2, TrReq3).insert(TrReq4) = ATrans3X(TrReq4, TrReq1, TrReq2)
ENDCLASS (* Trans3X*)

```

Step 4.4: Identifying an Account Transaction Hierarchy

The different types of account transactions are represented in the class hierarchy in figure 7.4.



The most important class in this hierarchy is `ServiceReq`. This is used to parameterise the `TRANSFORMER` attributes on the account database, and consequently simplify the specification. All the classes in the account transaction hierarchy are defined, in OO ACT ONE, below.

Step 4.5: Analysis and Synthesis of AccDB

The account database must provide a store for accounts and a means of accessing and updating the information associated with each account in the store. The analysis and synthesis of the account database class `AccDB` required many iterations of the analysis-synthesis-backtrack sequence. The header for the final version of the `AccDB` class is specified below.

All `AccDB` `TRANSFORMER` services are offered through one external attribute, namely `service`. The attribute `service` is then parameterised on an account identifier and transformer transaction (of class `ServiceReq`. This parameterisation helps to simplify the external interface of the `AccDB` class. Another important aspect of the `AccDB` class is that exceptions are specified when services are requested of accounts which are not in the account database. The hidden attribute `serviceOK`, used by the external attribute `service`, is defined (and called) only for services on valid accounts. This behaviour is specified by the equation definitions of the `AccDB` class, below.

An important aspect of the `AccDB` specification is the way in which changes to particular accounts in the database are made. The `TRANSFORMER` operations are defined by removing the identified account,

```

CLASS TrReq USING Sum OPNS STRUCTURES: DepositStr<Sum>,WithdrawStr<Sum> ENDCLASS
CLASS EnqReq OPNS LITERALS: StatementLit, AvailableLit ENDCLASS (*TrReq*)
CLASS BalanceEnq OPNS LITERALS: BalanceLit
ACCESSORS: isService -> Bool,isStatement -> Bool,isAvailable -> Bool ENDCLASS (*BalanceEnq*)
CLASS ChangeReq OPNS STRUCTURES: ChOverStr<Sum>, ChangeReqStr<Sum> ENDCLASS(* ChangeReq*)
CLASS Deposit USING TrReq SPECIALISES TrReq TO OPNS STRUCTURES: DepositStr
ACCESSORS: isService -> Bool, isStatement -> Bool, isAvailable -> Bool ENDCLASS (*Deposit*)
CLASS Withdraw USING TrReq SPECIALISES TrReq TO OPNS STRUCTURES: WithdrawStr
ACCESSORS: isService -> Bool, isStatement -> Bool, isAvailable -> Bool ENDCLASS (*Withdraw*)
CLASS Statement USING EndReq SPECIALISES EndReq TO OPNS LITERALS: StatementLit
ACCESSORS: isService -> Bool, isStatement -> Bool, isAvailable -> Bool ENDCLASS (*Statement*)
CLASS Available USING EndReq SPECIALISES EndReq TO OPNS LITERALS: AvailableLit
ACCESSORS: isService -> Bool, isStatement -> Bool, isAvailable -> Bool ENDCLASS (*Available*)
CLASS ChOver USING ChangeReq SPECIALISES ChangeReq TO OPNS STRUCTURES: ChOverStr
ACCESSORS: isService -> Bool, isStatement -> Bool, isAvailable -> Bool ENDCLASS (*ChOver*)
CLASS ChRest USING ChangeReq SPECIALISES ChangeReq TO OPNS STRUCTURES: ChRestStr
ACCESSORS: isService -> Bool, isStatement -> Bool, isAvailable -> Bool ENDCLASS (*ChRest*)
CLASS ServiceReq USING TrReq, ChangeReq GENERALISES TrReq, ChangeReq ENDCLASS (*ServiceReq*)
CLASS AccTrans USING ServiceReq,BalanceReq,ChangeReq
GENERALISES ServiceReq,BalanceReq,ChangeReq ENDCLASS (*AccTrans*)
(* The ACCESSOR equations are not given above: their definitions are intuitive *)

```

```

CLASS AccDB USING Acc, ServiceReq OPNS
LITERALS: emptyAccDB STRUCTURES: AccStr< Acc, AccDB >
ACCESSORS:available<AccID> -> Sum,statement<AccID> -> Trans3X,
getAcc<AccID> ->Acc(*HIDDEN*),isAcc<AccID>
->Bool(*HIDDEN*),zeroBalance<AccID>->Bool(*HIDDEN*)
TRANSFORMERS: newAcc<Acc>,delAcc<AccID>,service<AccID,ServiceReq>,
deposit<AccID,Sum> (*HIDDEN*), withdraw<AccID,Sum> (*HIDDEN*),
chOver<AccID,Sum>(*HIDDEN*),chRest<AccID,Sum>(*HIDDEN*),serviceOK<AccID,ServiceReq>(*HIDDEN*)
EQNS ...ENDCLASS (* AccDB *)

```

updating the account in an appropriate fashion and reconstructing the database by adding in the old account with its new state. The behaviour defined in this way is simple to understand but it should be clear that it is definitely not efficient. The analyst *must not* change the requirements model just because it is inefficient: the most important property of the requirements model is that the customer can understand it.

Step 5: Analysis of Transaction Class and Synthesis of Requirements Model

The `Transactions` class is used to store the account transaction messages, of class `TransMessage`, which are currently being routed to/from the target control node. It is defined as a recursive structure of `TransMessage` elements. It has an external interface composed from two attributes: a `TRANSFORMER` `add` and a `DUAL` `remove`. The class is defined below.

The behaviour of the `Transactions` class is simple to understand and easy to communicate with the customer. Consequently, *customer validation* of its behaviour is straightforward.

```

CLASS AccDB ... (* Header as above *) EQNS emptyAccDB.getAcc(AccID1) = ~Acc;
emptyAccDB.isAcc(AccID1) = false;
emptyAccDB.zeroBalance(AccID1) = ~Bool; emptyAccDB.available(AccID1) = ~Sum;
emptyAccDB.delAcc(AccID1) = ~AccDB;
(Acc1.getID).eq(AccID1) => AccStr(Acc1, AccDB1).getAcc(AccID1) = Acc1
OTHERWISE AccDB1.getAcc(AccID1);
(Acc1.getID).eq(AccID1) => AccStr(Acc1, AccDB1).isAcc(AccID1) = true
OTHERWISE AccDB1.isAcc(AccID1);
(Acc1.getID).eq(AccID1) => AccStr(Acc1, AccDB1).zeroBalance(AccID1) = Acc1.zeroBalance
OTHERWISE AccDB1.zeroBalance(AccID1);
(Acc1.getID).eq(AccID1) => AccStr(Acc1, AccDB1).available(AccID1) = Acc1.available
OTHERWISE AccDB1.available(AccID1);
AccDB1.getAcc(AccID1)..isBusiness => AccDB1..statement(AccID1) =
  AccDB1.getAcc(AccID1)..statement OTHERWISE ~Trans3X;
AccDB1.newAcc(Acc1) = AccStr(Acc1, AccDB1);
(Acc1.getID).eq(AccID1).not => AccStr(Acc1, AccDB1).delAcc(AccID1) =
  AccStr(Acc1, AccDB1.del(AccID1)) OTHERWISE (Acc1..balance).eq(0) => AccDB1 OTHERWISE ~AccDB;
AccDB1.getAcc(AccID1)..isRest => AccDB1..chRest(Acc, Sum1) =
  (AccDB1.delAcc(AccID1)).newAcc(AccDB1..getAcc(AccID1)..chRest(Sum1)) OTHERWISE ~AccDB1;
AccDB1.deposit(AccID1, Sum1) =
  (AccDB1.delAcc(AccID1)).newAcc(AccDB1..getAcc(AccID1)..deposit(Sum1));
AccDB1.withdraw(AccID1, Sum1) =
  (AccDB1.delAcc(AccID1)).newAcc(AccDB1..getAcc(AccID1)..withdraw(Sum1));
AccDB1.chOver(AccID1, Sum1) =
  (AccDB1.delAcc(AccID1)).newAcc(AccDB1..getAcc(AccID1)..chOver(Sum1));
AccDB1.isAcc(AccID1) => AccDB1.service(AccID1, ServiceReq1) =
  AccDB1.serviceOK(AccID1, ServiceReq1) OTHERWISE ~Acc;
AccDB1.serviceOK(AccID1, ChOverStr(Sum1)) = AccDB1.chOver(AccID1, Sum1);
AccDB1.serviceOK(AccID1, ChRestStr(Sum1)) = AccDB1.chRest(AccID1, Sum1);
AccDB1.serviceOK(AccID1, DepositStr(Sum1)) = AccDB1.deposit(AccID1, Sum1);
AccDB1.serviceOK(AccID1, WithdrawStr(Sum1)) = AccDB1.withdraw(AccID1, Sum1);
ENDCLASS (* AccDB *)

```

```

CLASS Transactions USING TransMessage OPNS
LITERALS: NoTrans STRUCTURES: TransStr< Trans, Transactions >
TRANSFORMERS: add<TransMessage> DUALS: remove<MessageID> -> TransMessage
EQNS Transactions1.add(TransMessage1) = TransStr(TransMessage1, Transactions1);
NoTrans.remove(MessageID1) = NoTrans;
(TransMessage1.getID).eq(MessageID1) =>
TransStr(TransMessage1, Transactions1).remove(MessageID1) = Transactions AND TransMessage1
OTHERWISE TransStr(TransMessage1, Transactions1).remove(MessageID1)
ENDCLASS (* Transactions *)

```

Step 5.1: Analysis of TransMessage Class and Synthesis of Requirements Model

The TransMessage class is defined as a fixed STRUCTURE of five components:

- A message identifier, which is defined as a renaming of ID. The LITERAL 0 is renamed M0 and the STRUCTURE IDStr is renamed M.

- An identification of the target control node, defined as a `NodeID`.
- An identification of the node which originated the transaction request, defined as a `NodeID`.
- An identification of the account to which the request is being sent, defined as an `AccID`.
- The request details, defined as an `AccTrans`.

The `TransMessage` class offers three external attributes:

- an ACCESSOR `getAccTrans` to return the `AccTrans` component.
- an ACCESSOR `getAccID` to return the `AccID` component.
- an ACCESSOR `getMessageID` to return the `MessageID` component.

The behaviour of class `TransMessage` is specified below.

```

CLASS TransMessage USING AccTrans, NodeID, MessageID OPNS
STRUCTURES: ATransMessage< MessageID, NodeID (*to*), NodeID (*from*), AccID, TrReq >
ACCESSORS: getMessageID ->MessageID, getAccTrans ->AccTrans, getAccID ->AccID
EQNS ATransMessage(MessageID1, NodeID1, NodeID2, AccID1, AccTrans1)..getMessageID = MessageID1;
ATransMessage(MessageID1, NodeID1, NodeID2, AccID1, AccTrans1)..getAccID = AccID1;
ATransMessage(MessageID1, NodeID1, NodeID2, AccID1, AccTrans1)..getAccTrans = AccTrans1
ENDCLASS (* TransMessage *)

```

Step 6: Specifying a class for returning enquiry results

A new class, `AccTransResult`, is required to represent the result of a transaction enquiry at the `BankingNetwork` interface. This class was not initially identified in the composition analysis because it is not a component of the `BankingNetwork`. However, it is now clear that such a class is required for the transfer of enquiry results across the network. `AccTransResult` is defined below.

```

CLASS AccTransResult USING Sum, Trans3X OPNS
STRUCTURES: StatementRes< Trans3X >, AvailableRes < Sum >
ENDCLASS (* AccTransResult *)

```

`AccTransResult` requires no external attributes because it is being used as a passive data representation. Classes which use the `BankingNetwork` can re-use `AccTransResult` and extend it with external attributes, if necessary.

Step 7: Synthesis of BankingNetwork Class

The classes which `BankingNetwork` depends on have been *customer validated*. The last step in the case study is the synthesis and analysis of the `BankingNetwork` class. Again, like its component classes, the `BankingNetwork` required many iterations of the analysis-synthesis-backtrack sequence before the requirements model was *customer validated*.

Step 7.1 Review of the BankingNetwork Component Classes

The **BankingNetwork** is composed from three component classes:

- **Network**: a network of nodes and links.
- **Transactions**: a set of internal message transactions which are in the process of being routed.
- **AccDB**: the data base of accounts which is the target for all the transactions in the system.

The *Interface O-LSTSD* for each of these classes is given in figure 7.5. The **BankingNetwork** provides its external functionality by delegating requests to these component classes.

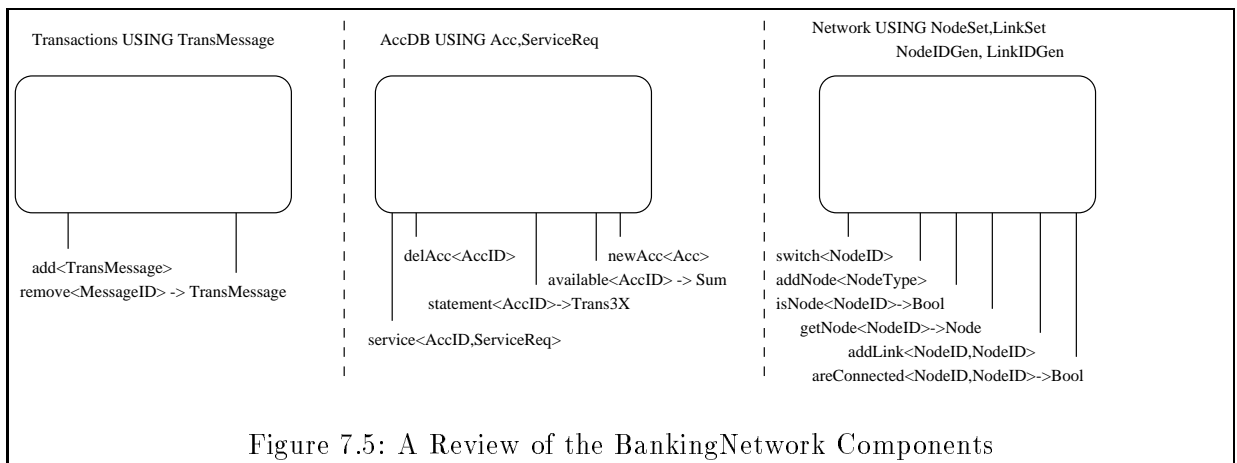


Figure 7.5: A Review of the BankingNetwork Components

Step 7.2 The External Interface of the BankingNetwork

The external interface of the **BankingNetwork** is defined by the class header, below.

```

CLASS BankingNetwork USING AccDB, AccTransResult, Network, Transactions OPNS
STRUCTURES: ABankNet< AccDB, Network, Transactions >
TRANSFORMERS: addNode<NodeID,NodeType>, addLink<NodeID,NodeID,NodeID>, switch<NodeID>,
accRequest<NodeID, TransMessage>, delAcc<NodeID, AccID>, newAcc<NodeID, Acc>,
arrived<MessageID>(*INTERNAL*), timeout<MessageID>(*INTERNAL*)
DUALS: returned<MessageID> -> AccTransResult (*INTERNAL*) EQNS ...

```

There are four important aspects of the **BankingNetwork** header definition:

- i) The **INTERNAL** transformers are used to model the communication aspects of the system:
 - The **arrived** service models a message transaction arriving at the appropriate control node, and results the account information being updated.
 - The **returned** service models the arrival of an enquiry reply at the node which originated the enquiry.
 - The **timeout** service models the nonservicing of a message transaction.
- ii) All the external attributes are defined as **TRANSFORMERS**. The enquiry services cannot be defined as **DUALS** because there is no guarantee that these requests will be serviced.

- iii) All the external attributes have a `NodeID` as their first parameter which is used to identify the node at which a transaction is originated. The `BankingNetwork` must ensure that only the correct sort of transactions are requested at particular types of node.
- iv) The `switch` attribute is interesting because the one `NodeID` parameter identifies both the node to be switched and the node at which the `switch` request is being made. In other words, `switch` requests cannot be routed across the network: they must occur at source.

The `BankingNetwork` class structure diagram is given in figure 7.6.

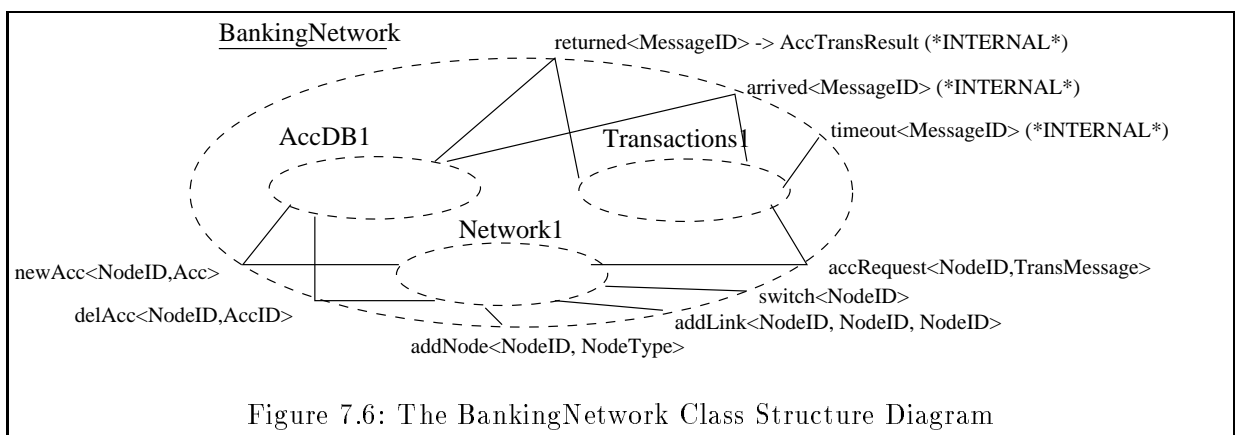


Figure 7.6: The `BankingNetwork` Class Structure Diagram

Step 7.3 Specification of `BankingNetwork` Behaviour

Informally, the external attributes of the `BankingNetwork` offer the following behaviour:

- `addNode<NodeID, NodeType>`:
The node identifier is checked, using the `Network` component, to guarantee that it corresponds to a control node in the system which is on: if so, a new node of type `NodeType` is added to the `Network`, otherwise the system ignores the request.
- `addLink<NodeID, NodeID, NodeID>`:
All three node identifiers are checked, using the `Network` component, to test that:
 - the first node identifier corresponds to a control node which is on.
 - the other two node identifiers correspond to different nodes in the `Network`

If so, a new link, joining the nodes identified by the second two `NodeID` parameters, is added to the `Network`, otherwise the system ignores the request.

- `newAcc<NodeID, Acc>`:
The node identifier is checked, using the `Network` component, to guarantee that it corresponds to a control node in the system which is on: if so, a new account (`Acc`) is added to the `AccDB`, otherwise the system ignores the request.
- `delAcc<NodeID, AccID>`:
The node identifier is checked, using the `Network` component, to guarantee that it corresponds

to a control node in the system which is on: if so, the account is checked to ensure that it has a zero balance, in which case it is deleted from the database, otherwise the system ignores the request.

- **switch<NodeID>:**

The **NodeID** is tested to verify that it corresponds to a node in the **Network**: if so, this node is switched, otherwise the system ignores the request.

- **accRequest<NodeID, TransMessage>:**

The **NodeID** is tested to verify that it corresponds to an ATM node which is on: if so, the **TransMessage** request is added to the system **Transactions** component for routing to the appropriate control node, otherwise the system ignores the request.

The **INTERNAL** attributes model nondeterministic state transitions:

- **timeout<MessageID>:**

This removes the specified message from the **Transactions** component of the **BankingNetwork**.

- **arrived<MessageID>:**

This removes the specified message from the **Transactions** component of the **BankingNetwork**. The message is then used to update the account database **AccDB**. The request is ignored if the message identifier does not correspond to a service request (deposit or withdraw).

- **returned<MessageID>:**

This removes the specified message from the **Transactions** component of the **BankingNetwork**. The message is then used to access the relevant account information in the account database **AccDB**, and this data is returned. The request is ignored if the message identifier does not correspond to an enquiry request (statement or available).

The **BankingNetwork** behaviour is formally defined in OO ACT ONE below.

The **BankingNetwork** is now ready for design.

7.2.3 A Review of the Analysis and Requirements Capture

The formal object oriented analysis, as performed in the case study, illustrates many of the main analysis and requirements capture issues:

- **Flexibility:** The need for an opportunistic method.
- **Executability:** The advantages of an executable model.
- **Customer Orientation:** The advantages of graphical notations.
- **Formality:** Abstraction and Nondeterminism.

Each of these issues is reviewed, with respect to the case study, in sections 7.2.3.1 to 7.2.3.4, below.

```

CLASS BankingNetwork (* Header above *) EQNS
(((Network1.getNode(NodeID1))..isControl).and((Network1.getNode(NodeID1))..isOn) =>
ABankNet(...).addNode(NodeID1,NodeType1) =
  ABankNet(AccDB1,Network1.addNode(NodeType1),Transactions1) OTHERWISE ABankNet(...);
(((Network1.getNode(NodeID1))..isControl).and((Network1.getNode(NodeID1))..isOn) =>
ABankNet(...).addLink(NodeID1, NodeID2, NodeID3) =
  ABankNet(AccDB1,Network1.addLink(NodeID2,NodeID3),Transactions1) OTHERWISE ABankNet(...);
ABankNet(...).switch(NodeID1) = ABankNet(AccDB1, Network1.switch(NodeID1), Transactions1);
(((Network1.getNode(NodeID1))..isControl).and((Network1.getNode(NodeID1))..isOn) =>
ABankNet(...).newAcc(NodeID1,Acc) =
  ABankNet(AccDB1.newAcc(Acc),Network1,Transactions1) OTHERWISE ABankNet(...);
(((Network1.getNode(NodeID1))..isControl).and((Network1.getNode(NodeID1))..isOn) =>
ABankNet(...).delAcc(NodeID1,AccID) =
  ABankNet(AccDB1.delAcc(AccID),Network1,Transactions1) OTHERWISE ABankNet(...);
(((Network1.getNode(NodeID1))..isATM).and((Network1.getNode(NodeID1))..isOn) =>
ABankNet(...).accRequest(NodeID1,TransMessage1) =
  ABankNet(AccDB1,Network1,Transactions1.add(TransMessage1)) OTHERWISE ABankNet(...);
ABankNet(...).timeout(NodeID1,MessageID1) =
ABankNet(AccDB1,Network1,Transactions1.remove(MessageID);
((Transactions1..remove(MessageID))..getAccTrans)..isStatement =>
ABankNet(...).returned(NodeID1,MessageID1) =
  ABankNet(AccDB1,Network1,Transactions1.remove(MessageID1)) AND
  StatementRes(AccDB1..statement((Transactions1..remove(MessageID))..getAccID))
  OTHERWISE ((Transactions1..remove(MessageID))..getAccTrans)..isAvailable =>
  ABankNet(AccDB1,Network1,Transactions1.remove(MessageID1)) AND
  AvailableRes(AccDB1..available((Transactions1..remove(MessageID))..getAccID))
  OTHERWISE ABankNet(...);
((Transactions1..remove(MessageID))..getAccTrans)..isService =>
ABankNet(AccDB1,Network1,Transactions1).arrived(MessageID1)=
  ABankNet(AccDB1.service((Transactions..remove(MessageID1))..getAccID,
                          (Transactions..remove(MessageID1))..getAccTrans1),
          Network1,Transactions1.remove(MessageID1)) OTHERWISE ABankNet(...)
ENDCLASS (* BankingNetwork *)

```

7.2.3.1 Flexibility

The case study illustrates how the requirements model was developed in an opportunistic fashion. The analysis was both *bottom-up* and *top-down*:

- *Bottom-up*

The **BankingNetwork** was analysed *bottom-up*. The classes which the **BankingNetwork** depends on, including its component classes, were specified and validated before the **BankingNetwork** behaviour was fully understood: they were developed to achieve this understanding.

- *Top-down*

The **Network** behaviour was analysed *top-down*. The classes it depends on were specified and validated after the **Network** behaviour was understood: they were developed to record this

understanding.

In general, it is not possible, given a collection of classes requiring specification, to predetermine the *best* order in which these classes should be analysed, synthesised and validated. Customers and analysts must be encouraged to develop the requirements models as they see fit at the time. Opportunistic approaches are more difficult to organise (the size of the case study does not fully illustrate this) and so it is important that the requirements capture process is supported by management tools. Providing such support is beyond the scope of this thesis.

7.2.3.2 Executability

The OO ACT ONE requirements models are executable. This is very important in the process of *customer validation*. It is very difficult to communicate dynamic properties of a model if the requirements cannot be executed. Executing the requirements models helps the analyst to test their model against the behaviour they think the customer requires. It also helps the customer to validate the analyst's model against their requirements. Further, an executable model helps the customer and analyst to explore behaviour which is not well understood.

The development of the **BankingNetwork** requirements model involved many executions of the system class (and classes which the system depended on). Most of the backtracking took place in response to misunderstandings in the requirements model being identified during execution. It is unlikely that such problems would have been identified before the system was subsequently implemented. Only after the implementation was tested by the customer would these errors become evident. Executable requirements models reduce the risk of carrying errors into the design and implementation stages of software development.

Executable models are more costly to produce, but have the potential to reduce the expense of correcting errors during design and implementation. Further, the executable OO ACT ONE model is directly re-used in the object oriented designs. Consequently, there is no sense of losing work when the analysis is complete and design begins.

7.2.3.3 Customer Orientation

The OO ACT ONE requirements models are *customer oriented*. The **BankingNetwork** behaviour can be presented to the customer in a number of different forms:

- The dynamic model can be presented as sequences of state transitions. Event diagrams (see 3.5.6) provide a *customer oriented* view of system executions.
- The static properties, namely classification, subclassing, composition, dependency and configuration, can be presented in class structure diagrams and subclassing hierarchy diagrams. These graphical views were prominent in the **BankingNetwork** case study.
- The object oriented framework provides a consistent approach for the analyst to develop mutual understanding with the customer. The customer does not have to change conceptual frameworks when going between their understanding of the requirements and their understanding of the

requirements model: the object oriented style of specification means that these *should* be the same. Analysts must build the requirements model for the customer. This *customer awareness* is evident in the **BankingNetwork**: for example, we questioned the way in which the **Network** behaviour was specified.

Although the graphical views of the requirements were prominent in the case study, it became clear that the synthesis, analysis and validation of behaviour would benefit greatly from comprehensive tool support: all models and views were generated by hand. The ability to make quick changes to a requirements model was not matched by a similarly quick means of presenting the graphical views. This is an area of further work.

7.2.3.4 Abstraction and Nondeterminism

Two main types of abstraction are evident in the formal requirements model:

- Functional abstraction — every class of behaviour can be treated as an interface of well-defined (and well understood) external attributes.
- Exceptions — when a customer wants some sort of behaviour to be coped with in the final implementation, but is not yet willing (or able) to be more precise then exceptions are useful abstraction mechanisms.

Nondeterminism is a very important part of most requirements models. In the case study non-determinism (in the form of **INTERNAL** state transitions) was used to model the internal routing of messages. Some messages are routed correctly whilst others are lost (timed out): *how* this behaviour is defined is not specified. It is the role of the analyst to communicate the nondeterministic aspects of the requirements model with the designers. It is the designers who must remove the nondeterminism.

7.3 Design: Moving the System from Abstract to Concrete

This section reviews the process by which the OO ACT ONE **BankingNetwork** requirements were designed for implementation in Eiffel. Design proceeded in distinct steps. At the end of each step the new design was verified against the old design: when a correctness preserving transformation (CPT) was used then this verification was immediate. However, some of the design steps were not applications of pre-defined CPTs. In these cases, we either: formally verified the design step with a proof of correctness, or informally justified the design step as being correctness preserving and tested our reasoning by executing the LOTOS specification of the new design.

It is intended, in the future development of FOOD, that the OO LOTOS designs are hidden beneath some high-level object oriented design interface. The case study, however, required direct manipulation of the OO LOTOS code. At certain key stages in this section, LOTOS code fragments are presented. When appropriate, diagrammatic representations of the design (and design components) are given.

The underlying **BankingNetwork** functionality is contained within the ACT ONE implementation of the OO ACT ONE requirements model. This ACT ONE code is not reviewed as part of the thesis. In some instances, new classes of behaviour are required in the design: these classes are coded in OO ACT ONE and then translated to ACT ONE for use in the design.

The case study does not make use of all the CPTs, which are defined in chapter 5. Design is targeted towards a non-concurrent implementation (in Eiffel). As such, the transformations concerned with concurrency and distribution are not illustrated by the case study. However, there is reason to believe that the **BankingNetwork** model could be designed towards a concurrent implementation. An investigation of such a development is beyond the scope of this thesis.

It is not possible to examine, within this thesis, all aspects of the design of the **BankingNetwork** system. The main design steps are reviewed in sections 7.3.1 to 7.3.6. A review of the design process is given in section 7.3.7.

7.3.1 From Analysis to Design: Choosing the Communication Model

Eiffel is the target implementation language and so, for reasons given in chapter 5, the first design decision is to choose the remote procedure call (RPC) model of communication for the **BankingNetwork** system. The initial OO LOTOS design of the **BankingNetwork** is given below in the **PBankingNetwork1** process definition¹⁰.

```

process PBankingNetwork1[newAcc,delAcc,addNode,addLink,switch,accRequest]
(SBankingNetwork: BankingNetwork): noexit:= hide returned, arrived, timeout in
(* EXTERNAL INTERFACE *)
(newAcc?NodeID1:NodeID?Acc1:Acc;
PBankingNetwork[...](.newAcc( SBankingNetwork, NodeID1, Acc1)))
)[] ...[]
(* INTERNAL TRANSITIONS *)
(returned?MessageID1:MessageID?;
returned!AccTransResultResult(returned(SBankingNetwork, MessageID));
PBankingNetwork[...](.newAcc( SBankingNetwork, NodeID1, Acc1)))
)[] ...endproc (* PBankingNetwork1 *)

```

This initial design facilitates a review of syntactic conventions in the OO LOTOS designs:

- *Classes* from the requirements model are defined as *PClass* processes in the LOTOS designs. The version of the class definition is identified by a numeral at the end of the process identifier. As design progresses we verify that correctness is preserved from one version of the process definition to the next.
- The ACT ONE sorts, corresponding to classes in the requirements model, retain the class names used in the analysis. Class variables are represented by the class name preceded by an **S** (for state).

¹⁰The OO LOTOS process definitions that follow are not given in standard LOTOS syntax. For the sake of brevity, lists of gates, choices and operation parameters are often presented as `listElement1, ...listElement2`, when the context in which this syntax is used makes the identification of the complete list immediate and unambiguous.

- The external attributes of the requirements model classes have two correspondences in the LOTOS designs: as gate names in the process algebra and operation names in the ADT part.
- The ACT ONE operations `.(req(object, ...))` and `ClassResult(req(object, ...))` are used, respectively, to define the new state of an `object` after servicing a `req`, and the result, if any, returned from servicing a `req`.

7.3.2 Decomposition of the Banking Network System

The initial design is a more concrete implementation of the requirements model since it specifies *how* a `BankingNetwork` communicates with its environment through its external interface. The OO ACT ONE requirements are contained within the ADT part of the specification, and used directly in specifying *what* behaviour is offered at the interface. The initial design does not state *how* the `BankingNetwork` offers this behaviour: it is the designers who must specify *how* this behaviour is to be implemented.

A first step in the design process is to transfer the structure in the requirements into structure in the design: this structure is then amenable to manipulation. The `BankingNetwork` has a static structure and so we apply the static expansion CPT *StExp* to the `PBankingNetwork1` process definition. The second `BankingNetwork` design, resulting from the application of *StExp*, is defined as process `PBankingNetwork2`, below.

```

process PBankingNetwork2[newAcc,delAcc,addNode,addLink,switch,accRequest]
(SBankingNetwork: BankingNetwork): noexit:=
hide returned, arrived, timeout, Transactions1add, Transactions1remove, AccDB1service,
AccDB1delAcc, AccDB1newAcc, AccDB1statement, AccDB1available, Network1switch,
Network1addNode, Network1isNode, Network1getNode, Network1addLink, Network1areConnected in
BankingNetwork2Control[newAcc, ..., Network1areConnected]
|[ Transactions1add, ..., Network1areConnected] |]
( PTransactions[Transactions1add, Transactions1remove] (par1(SBankingNetwork)) |||
PAccDB[AccDB1service, ..., AccDB1available] (par2(SBankingNetwork)) |||
PNetwork[ Network1switch, ..., Network1areConnected] (par3(SBankingNetwork))
)endproc (* PBankingNetwork2 *)

```

The `BankingNetwork2` component processes `PAccDB`, `PTransactions` and `PNetwork` are generated as RPC-model designs from their OO ACT ONE specifications. The `BankingNetwork2Control` process co-ordinates the way in which these components are used to provide the external behaviour of `BankingNetwork2`. It is defined below.

The second version of the design is represented in the diagram in figure 7.7, which shows quite effectively the result of the static expansion. Henceforth, similar diagrams are used to illustrate the process structure in the OO LOTOS designs. LOTOS text is included only where necessary.

7.3.3 Decomposition of the Network Component Process

The next design step is the static expansion of the `PNetwork1` process. The result of applying *StExp* to `PNetwork1` is illustrated in figure 7.8. This expansion is necessary because we intend to change the

```

process BankingNetwork2Control[ newAcc,..., Network1areConnected]: noexit:=
(newAcc?NodeID1:NodeID?Acc1:Acc;
Network1getNode!NodeID1?Node1:Node;
( [and(BoolResult(isControl(Node1), BoolResult(isOn(Node1))))] ->
AccDB1newAcc!Acc1; BankingNetwork2Control[...] )
[] (
[not(and(BoolResult(isControl(Node1), BoolResult(isOn(Node1))))] -> BankingNetwork2Control[...]
))][...endproc (* BankingNetwork2Control *)
    
```

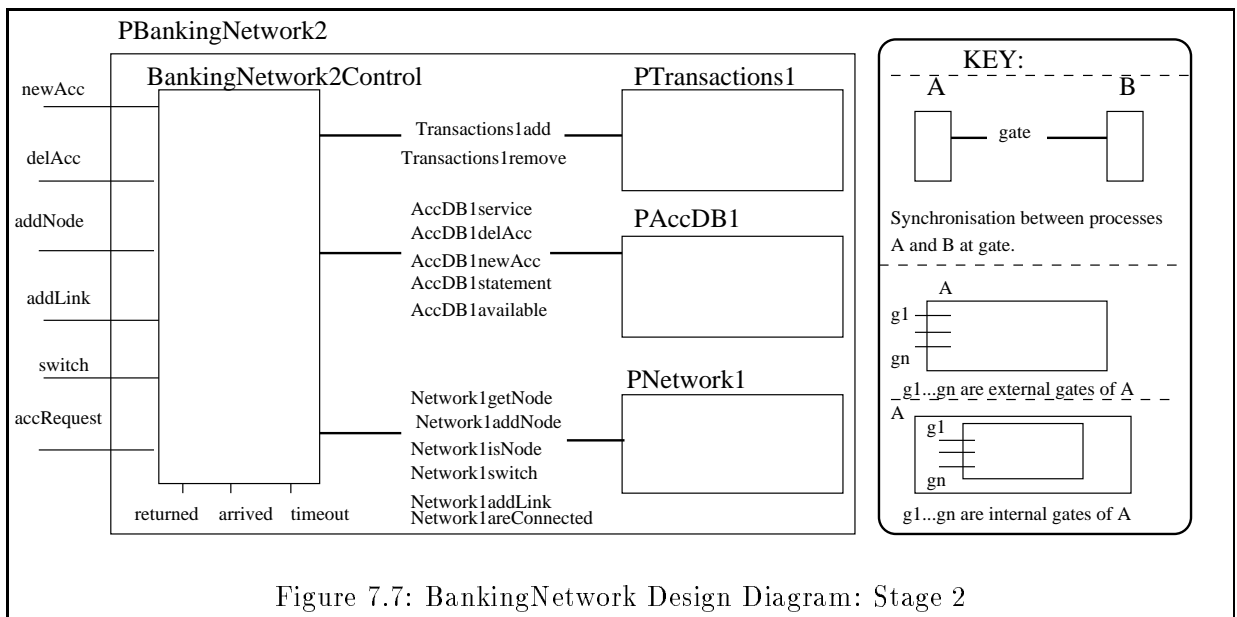


Figure 7.7: BankingNetwork Design Diagram: Stage 2

internal structure of the **Network**.

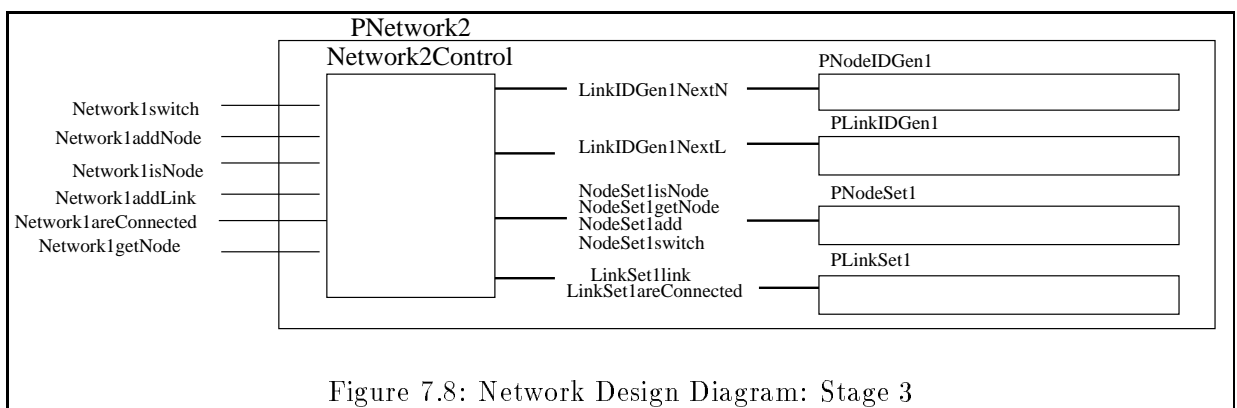


Figure 7.8: Network Design Diagram: Stage 3

7.3.4 Restructuring the Network Component Process

The purpose of the restructuring is to change the way in which **Network** topology is defined. Rather than having a distinct set of **Links**, all the **LinkSet** elements are to be distributed between the **Nodes** in the **NodeSet**. This is achieved in two steps:

- i) Compose the `NodeSet` and the `LinkSet` in the `Network`, using the `Comp` CPT.
- ii) Integrate the two recursive structures in the ADT parts of the `NodeSet` and `LinkSet` into one recursive structure of `LinkedNodes` in a `LinkedNodeSet`.

7.3.4.1 Composing NodeSet and LinkSet

The next version of the `Network` design is defined as a composition of the `NodeSet` and `LinkSet` components: $PNetwork3 = Comp(PNetwork2, \{\{1\}, \{2\}, \{3,4\}\})$. This new structure is illustrated in figure 7.9.

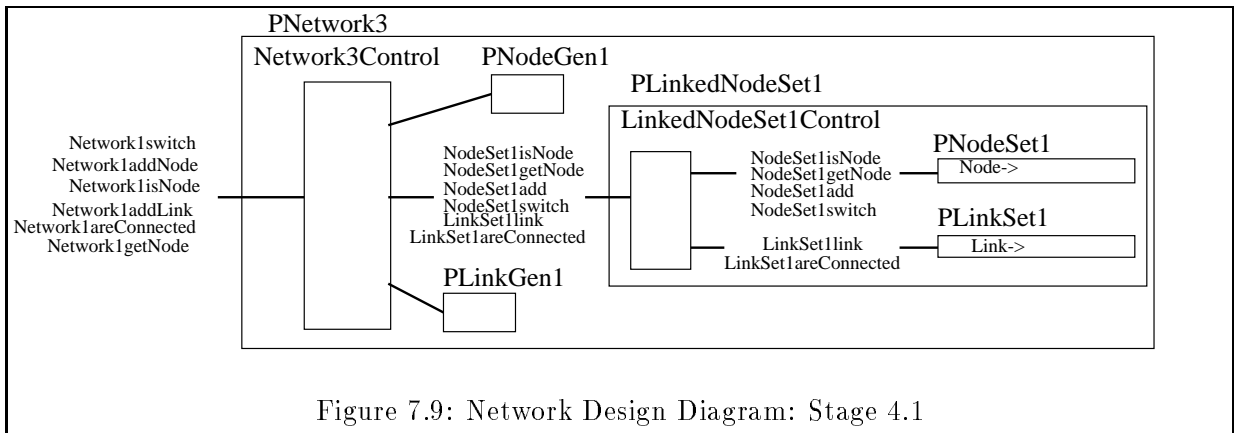


Figure 7.9: Network Design Diagram: Stage 4.1

The initial composition is important because it permits manipulation of the `LinkedNodeSet` specification, independent of the other `Network` components.

7.3.4.2 Merging the NodeSet and the LinkSet

A CPT for the merging of two recursive structures has not been developed as part of this thesis. Consequently, we must examine the transformation in some detail in order to be sure of its correctness.

The idea behind the transformation is very simple: every component of the `LinkSet` should be placed somewhere in the `NodeSet`. The simplest way of achieving this is to give each `Node` in the `NodeSet` an extra component: a `LinkSet`. Then, this extra component can be used to store a subset of the `Network`'s `LinkSet`. We require that every `Link` in the `Network LinkSet` is represented in the new set of 'Nodes with Links', named `LinkedNodeSet`. The `LinkedNodeSet` is defined as a recursive structure of `LinkedNode` elements. It is this new set which defines version 2 of the `PLinkedNodeSet` process definition. This new process is illustrated in figure 7.10.

7.3.5 Integrating the Transaction Set in the Network

The mechanism used to merge the `LinkSet` and `NodeSet` is re-applied to merge `Transactions`, a set of `Transaction` components, with the `LinkedNodeSet`. This merging requires an initial restructuring of the `BankingNetwork`, as illustrated in figure 7.11.

The merging requires a new `Transactions` component to be added to the `LinkedNode` structure to define a new class, namely `TransLinkNode`. The `LinkedNodeSet` and `Transactions` components

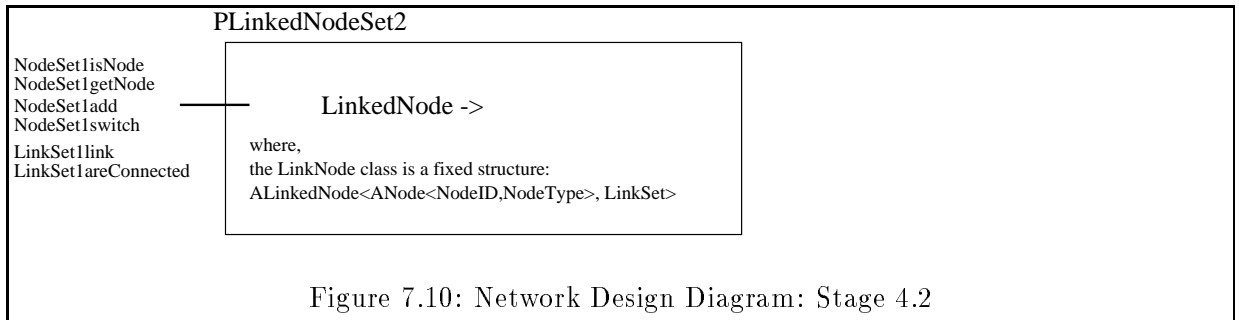


Figure 7.10: Network Design Diagram: Stage 4.2

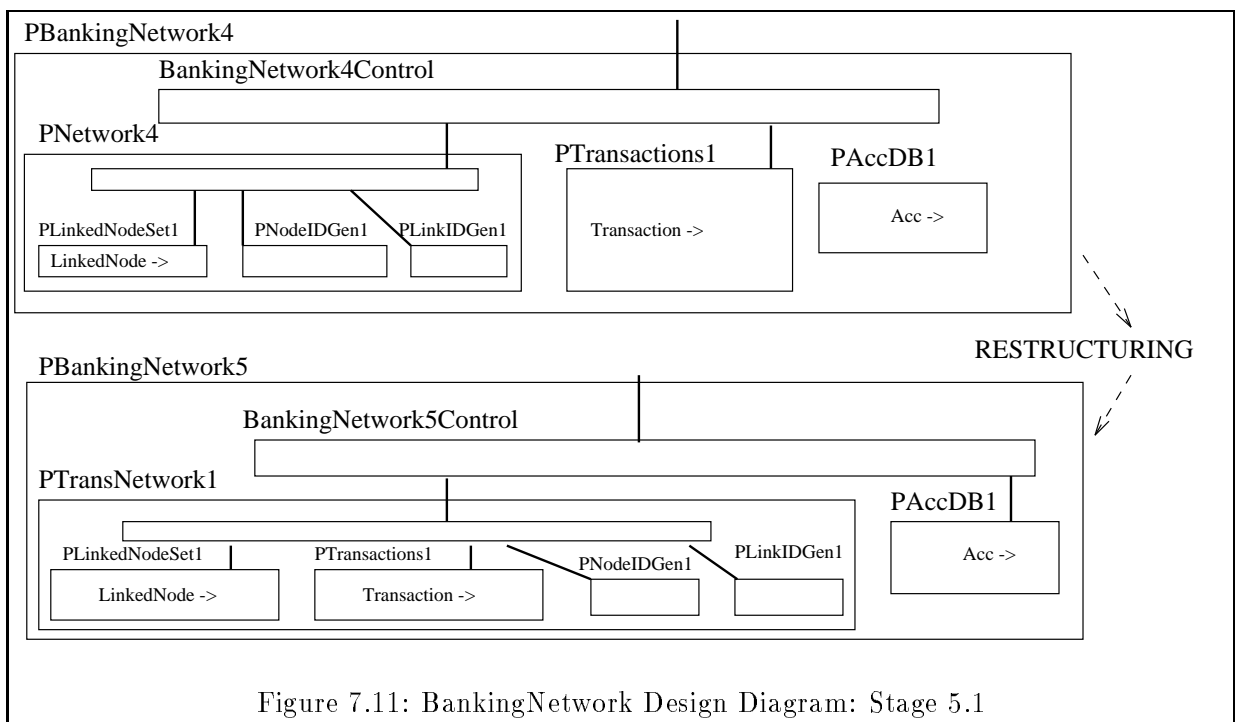


Figure 7.11: BankingNetwork Design Diagram: Stage 5.1

are merged into one component (a `TransLinkNodeSet`) which is defined as a recursive structure of `TransLinkNode` elements. Every `Transaction` in the `BankingNetwork` is recorded in one, and only one, of the `TransLinkNodes` (as a member of its `Transactions` component). The `BankingNetwork` design, after the merging of `Transactions` and the `Network` components, is illustrated in figure 7.12.

7.3.6 An Explicit Routing Mechanism: Removing Nondeterminism

The `BankingNetwork` design has been transformed with the intention of explicitly specifying an internal mechanism for the routing of transaction messages. The `TransLinkNodeSet` class is ready for modelling the routing of messages between nodes. This is done in two stages:

- i) Modelling the movement of messages along links.
- ii) Designing a routing mechanism as a means for nodes to determine on which link outgoing messages are to be sent.

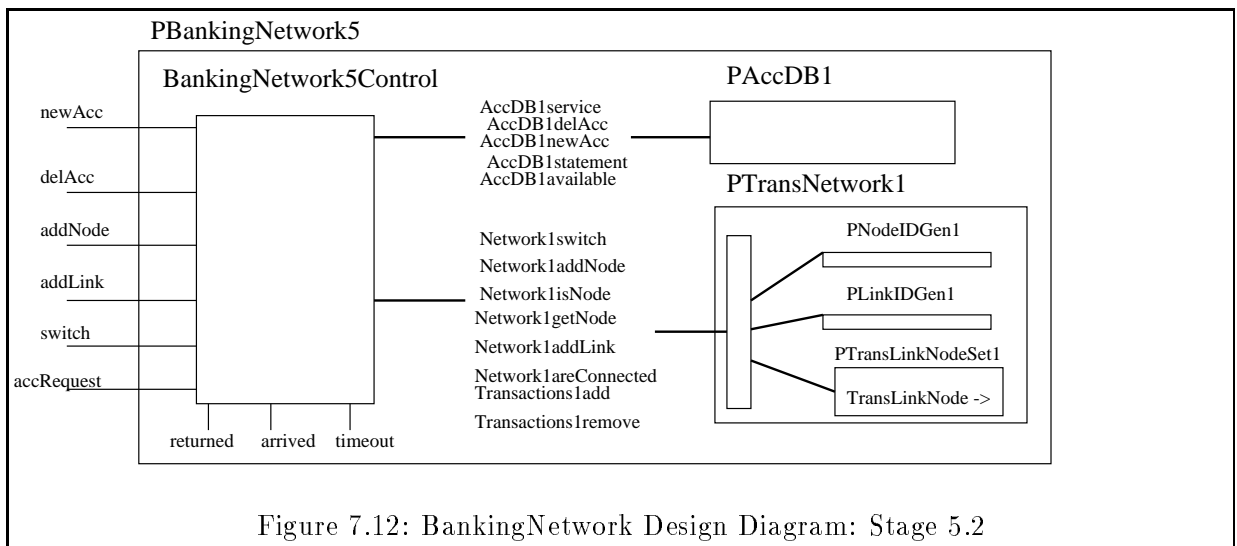


Figure 7.12: BankingNetwork Design Diagram: Stage 5.2

7.3.6.1 Modelling the internal movement of messages

A new internal transition $\text{move}\langle \text{NodeID}, \text{NodeID} \rangle$ is defined to model the movement of a message transaction from the first node to the second (as specified by the `NodeID` parameters). The `TransLinkNode` identified by the first `NodeID` must decide which of its currently held transactions is sent across the link to the node specified by the second `NodeID` parameter. This is determined by the routing mechanism. When a message is received by a `TransLinkNode` then one of four things occurs:

- i) The message is a service request which can be serviced by the receiving (control) node. As a result, the `AccDB` is updated appropriately and the message is removed from the network. This models the internal `arrived` transaction.
- ii) The message is an enquiry which can be serviced by the receiving (control) node. As a result, the `AccDB` is accessed to obtain the appropriate reply to the enquiry, the original message is removed from the network and a new message (carrying the reply) is added to the network to be routed back to the node which originated the enquiry.
- iii) The message is a reply which has arrived back at the node which originated the enquiry. The information is given to the receiving node and the message is removed from the network. This models the `returned` transaction.
- iv) The message cannot be serviced by the receiving node so it is forwarded for routing to another node.

The internal `timeout` transformation is defined to remove the specified message from within whichever `TransLinkNode` it is stored.

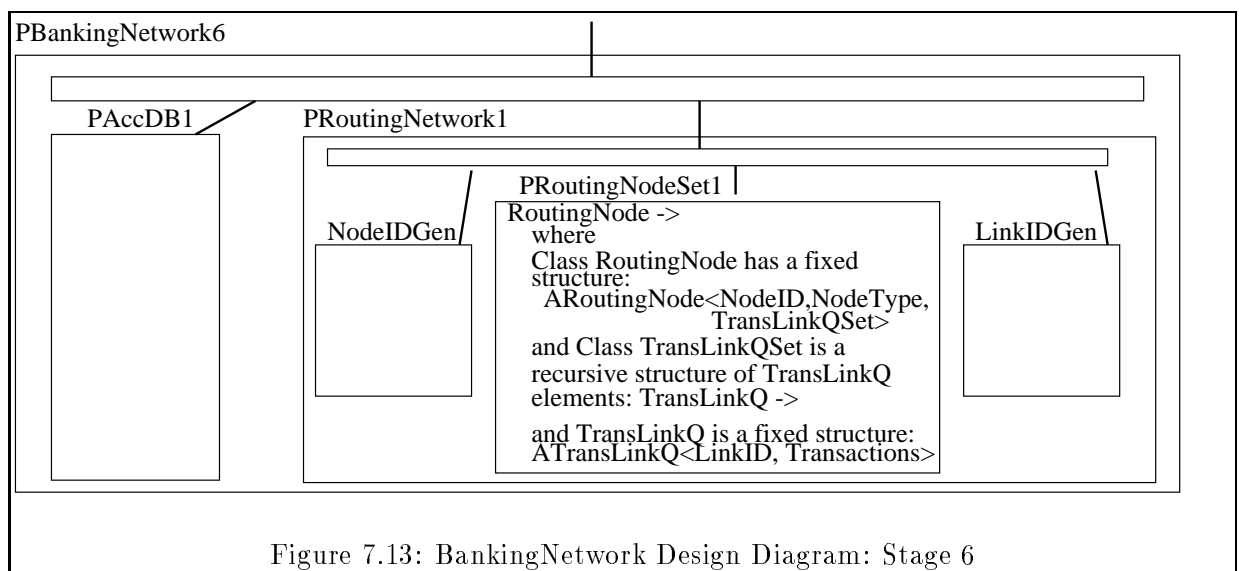
7.3.6.2 Designing A Routing Mechanism

The design needs to be extended to provide each `TransNode` with a routing mechanism. [55] examines a number of different network routing mechanisms: their specification in LOTOS and resulting

implementation in C++. In the **BankingNetwork** case study, one of the simplest routing mechanisms was chosen for implementation, namely the *Hot Potato* mechanism, see [102].

The *Hot Potato* routing algorithm is developed on the philosophy that it is always best to get rid of an incoming message (that needs routing) as quickly as possible. To model this, we define every **TransLinkNode** to have an additional component, namely a set of transition queues (one queue for each outgoing link in the node). Consequently, the **TransLinkNodeSet** of **TransLinkNode** components is replaced by a **RoutingNodeSet** of **RoutingNode** components. The movement of messages between nodes is simply achieved by popping a transaction off the queue, identified by a **LinkID**.

The specification of *Hot Potato* routing behaviour (and a number of extensions and refinements) is given in [55]. This behaviour is incorporated in the **BankingNetwork** in a straightforward manner (many of the classes are directly re-used). The final design of the **BankingNetwork** is illustrated in figure 7.13. It is this design which is put forward for implementation in Eiffel.



7.3.7 A Review of the Design Process

7.3.7.1 Limitations of the RPC-model of communication

Designing towards a non-concurrent implementation is much simpler than designing towards a concurrent implementation. With an RPC-model of communication, the design decisions are primarily concerned with:

- Composition and decomposition of data structure.
- Removing nondeterminism.

Unfortunately, the CPTs which address concurrency and distribution design decisions are not relevant in a non-concurrent model. It is these issues which make the transformation to full LOTOS from OO ACT ONE a vital part of FOOD. This is not fully illustrated by the case study.

7.3.7.2 Towards A Concurrent Implementation

The case study addresses a problem which is inherently concurrent: the `BankingNetwork` is modelling a system of distributed `Node` processes. This concurrency is designed out of the system by introducing internal transformations for modelling the routing of messages between nodes. Given a concurrent target implementation language, the FOOD approach can target the design using CPTs, defined in chapter 5, which were not utilised in the case study. An area of future work is the investigation of the effectiveness of these other CPTs. A concurrent implementation of the `BankingNetwork` provides a suitable problem for such an investigation.

7.3.7.3 The Need For Tool Support

This thesis is principally concerned with mathematical models, i.e. conceptual tools. The CPT-driven design is particularly rich in this respect. Conceptual tools must be supported by development environments. The LOTOS tools (both the SEDOS tools and LITE tools were used) provided a limited support during the design stages of FOOD. Unfortunately, these tools do not support object oriented design: they support the development of LOTOS specifications. It is clear that the design stages of FOOD need more suitable tool support.

7.3.7.4 The Need For More CPTs

The case study illustrates the importance of CPTs in design. It is vital that a comprehensive set of object oriented design CPTs are developed. This will improve the quality of design and the productivity of designers. In short, CPTs allow designers to decide *what* should be done, rather than *how* it can be done.

This thesis introduces only a small number of CPTs. These are used to show that a CPT-driven approach to object oriented design is possible. However, the case study shows that many more CPTs are needed: for example, the merging of two recursive structures needs to be treated formally. It is hoped that CPTs become as widely re-used as classes of behaviour.

7.4 The Eiffel Implementation

It is not effective to examine the Eiffel code for the `BankingNetwork` (this code is available on request). Rather, an overview is given of the implementation process.

7.4.1 The Role of the Final Object Oriented LOTOS Design

After the development of the requirements model, and its manipulation in the design, the Eiffel implementation went as outlined in section 6.4. The final design had a direct influence on implementation since there was a correspondence between:

- Classes in the design and classes in the implementation.
- Class hierarchies in the design and class hierarchies in the implementation.

- Composition structure in the design and composition structure in the implementation.

7.4.2 Re-Use in the Implementation

The extensive library of Eiffel classes meant that many of the design classes were already implemented:

- The `Sum` class (in the account database) was implemented as a real number in Eiffel.
- The `TransLinkQ` class (in the network) was implemented as an instance of the generic queue class in Eiffel.
- The different identifier classes were all implemented as Eiffel integers.
- The recursive structures were implemented as Eiffel linked lists.

7.4.3 Implementing Exceptions

The exceptions in the requirements model were not designed out. Eiffel, as the target implementation language, provides an exception handling construct. Consequently, design did not have to address this behaviour. The handling of exceptions was straightforward: warning messages were returned to the user interface and the state of the system before the exception was returned to.

7.4.4 Implementing A User Interface

Perhaps surprisingly, the coding of the user-interface required more time than the coding of the functionality being provided at the interface. Initially, a graphical user interface to the `BankingNetwork` was developed in Eiffel. However, after struggling with many of the problems in the Eiffel class library, a simple textual interface was coded instead.

7.5 A Review of the Case Study

7.5.1 Development Statistics

An important aspect of the case study is the way in which the development effort was distributed between the three main stages of FOOD: requirements capture, design and implementation. Although it is difficult to be precise about development costs, the following statistics may be useful:

- **Requirements Capture**

Analysis and requirements capture required approximately one man month of work. It resulted in 1200 lines of OO ACT ONE code, which was translated into 6500 lines of ACT ONE. The requirements model contained 25 classes of behaviour, with an average of 6 external attributes per class. Over one half of the classes had a fixed structure.

- **Design**

Design required approximately three man weeks of work. It resulted in 1400 lines of LOTOS process algebra together with 7400 lines of ACT ONE. Design introduced 8 new classes of behaviour.

- **Implementation**

Implementation required approximately one man month of work (of which more than half was spent developing a user-interface). The Eiffel code (ignoring the user interface) was approximately 800 lines. The reduction in code was a consequence of three things:

- The OO LOTOS design had to explicitly define the communication model, whilst this is explicit in the Eiffel semantics.
- The reference semantics of Eiffel reduced the amount of code needed to define the routing of messages between nodes.
- The extensive use of pre-coded components.

The above statistics strengthen our claim that FOOD places much more emphasis on the earlier stages of development. This claim can be properly verified only if FOOD is used in a wide range of software engineering projects.

7.5.2 The Effectiveness of FOOD

The case study shows the effectiveness of FOOD arising out of the combination of formal and object oriented methods:

- **Formality**

A formal approach improved understanding, removed potential errors earlier in the development and made design decisions explicit.

- **Object Orientedness**

An object oriented approach provided a conceptual integrity, facilitated re-use at all stages of development and supported opportunistic development.

7.5.3 Extensions to the Behaviour

To illustrate the way in which FOOD supports extensions to systems that have already been developed, two extensions were considered: changing the routing mechanisms and providing more banking services.

7.5.3.1 Changing Routing Mechanisms

The `BankingNetwork` incorporated a very simple routing mechanism: the hot potato algorithm. Two other, more complex routing mechanisms replaced the hot potato method. These new mechanisms were integrated into the system by first making changes to the design and then updating the Eiffel code:

- **A Flooding Mechanism**

In this model, incoming messages are sent on all outgoing links and a constraint was introduced so that messages were moved only a limited number of times. This required approximately one man week of work.

- **A Backward Learning Mechanism**

Each node was extended to incorporate a store of the shortest paths to other nodes. This store is updated when an incoming message has come by a route which was shorter than that stored. Consequently, every message must carry additional routing information. This change required approximately three man weeks of work. The additional routing behaviour was very complex and so, as a preliminary to design, OO ACT ONE was used to construct a routing requirements model. This improved our understanding of the behaviour required and the design involved making decisions as to how this behaviour was to be implemented.

7.5.3.2 Providing Additional Banking Functionality

The facilities provided for each bank account were extended to include a mechanism for transferring money from one account to another. This extension was carried through by first making changes to the requirements model and then proceeding with design and implementation. The updated bank accounts were defined as subclass (extensions) of the original classes and the design structure needed minimal changes. This whole process took less than one man day.

7.5.3.3 Lessons Learned From Extensions

Four important lessons were learned from the case study:

- Re-use through composition is much more common than re-use through inheritance.
- When building a system it is important to keep potential extensions in mind.
- Meaningful generalisation should be applied when possible: the extra development time is rewarded with benefits in later projects.
- Components that can be extended within one application area are not necessarily re-usable in other application areas: for example, it is unlikely that the routing mechanisms can be usefully re-used in other problem domains.

7.5.4 The Importance of Structure Throughout Development

The case study places emphasis on structure at all stages of development. The advantages of maintaining structure from specification to implementation are as follows:

- Traceability, in the sense of a design audit, is improved. Testing the implementation can be done in a constructive fashion
- Extending or changing a system can be achieved in a controlled fashion. With an object oriented approach, modifications are often localised (resulting in changes to only a few classes in the system).
- Structure provides the framework upon which mutual understanding, between different members of the software development team, is based.