Formal Object-Based Design In LOTOS

J. Paul Gibson
Department of Computing Science,
University of Stirling,
Stirling FK9 4LA.
(jpg@uk.ac.stir.cs)

27th April 1991

Abstract

The advantages of object-based techniques in Software Engineering are well recognised and widely accepted. The inherent flexibility of object-based systems often leads to an interactive and incremental style of development, a consequence of which may be insufficient rigour in software production. As the size and complexity of a software project increases, it is important to adhere to as rigorous a method as possible.

This paper shows how a formal method can be incorporated into an object-based development strategy. In this way it is possible to provide all the advantages of the object-based paradigm whilst ensuring that the quality of code is not compromised. The specification language LOTOS is shown to be appropriate for the generation and evolution of formal object-based designs. The structure of the object-based specification, as a *design*, provides a foundation upon which an implementation can be constructed. A simple network routing system illustrates the role of the *design* in the object-based development process, from the generation of the LOTOS specification to the coding of the C++ implementation.

1 Introduction

Formal methods have traditionally been associated (in the Software Engineering field) with the initial stages of the software life cycle [7]. Specification languages act as a bridging mechanism among requirements analysis and system design. The object-based paradigm seems to provide a basis for conceptual integrity between all stages of the (software) development process [3]. The use of object-based techniques in a formal method therefore deserves investigation. Consistency between specification of the requirements and design of a solution has many potential benefits.

This paper does not propose a comprehensive object-based design method. The problem domain structure is used only as the basis for the design. Even in an object-based development method, it is unlikely that there is a complete correspondence between the solution domain structure and problem domain structure. An object-based LOTOS [8] specification does provide a design framework which may be formally transformed[1] to reflect architectural choice outside the problem domain. This paper shows how the structure of an object-based LOTOS specification corresponds to a particular design. The object-based paradigm guarantees some degree of consistency between the structures of the problem and solution domains.

This paper reports on a case study that used the structure of a LOTOS specification as an object-based design from which a C++ [11] implementation was developed. The study showed that such a development strategy is not only possible but also desirable. Furthermore, our experience with the case study illustrates how an object-based system can be evolved[2] controllably to more complex behaviour within a formal framework.

[1]The relationship between formal transformations and informal design decisions is outside the scope of the work. However, each design stage will have a corresponding object-based LOTOS specification.

[2]The evolution of design is a consequence solely of an extension to requirements. Changing the structure of the specification as a means of reconceptualising the problem is not considered in this paper.

2 Object-Based LOTOS

Specifications in LOTOS can be written in a number of different styles [14]. The object-based style [2] is supported by LOTOS [3].

2.1 Language Considerations

The following scheme informally relates LOTOS concepts to object-based terminology. Consistent adherence to this mapping forms the basis of an object-based style for LOTOS.

2.1.1 Process Algebra

- **System design is an aggregation of objects**
  The behaviour of a system can be defined as a number of parallel processes. Each of these processes can themselves be defined as an aggregation of component processes, and so on. **Process abstractions** are used to specify behaviours of systems (and subsystems) and correspond to the notion of **class**. **Process instantiations** correspond to the object-based notion of an **object**. In an instantiation, the process state has been allocated particular values and the process is at a particular point in its behaviour.

- **Objects interact at well defined interaction points**
  Object interaction is achieved through process synchronisation (which occurs at explicitly defined gates). **Process synchronisation** is used to establish or negotiate values, and this provides the basis for modelling **message-passing.** This relationship needs to be more fully developed because event synchronisation in LOTOS is not explicitly directional and may occur among more than two processes. We propose that the following convention is followed when using the object-based style. Each gate in the gate list of a LOTOS process should have an implicit direction (as either a *sender* or *receiver* of messages). The process, acting as an autonomous object, does not need to know what

[3]It is similar to resource-oriented style [13] in that there is a correspondence between objects and resources, but there are differences in the way that the process synchronisation mechanism is used.

it is interacting with. In this way, the LOTOS specification can contain multiprocess synchronisation. However, to ensure that there are no constraints imposed by the process on its environment (which may be other processes in the specification), an additional rule is required. Each process must always be able to participate in an event that occurs at a *receiving* gate.

- **Object-Based System Structure results from the construction of process definitions**
  In object-based LOTOS, the parallel composition operator is used in process definitions to define the behaviour of one process in terms of two or more others. Internal interaction between the components of an object (LOTOS process) is specified by hiding appropriate gates in the specification of that process. It is precisely these compositional aspects of the specification which represent the *design*.

- **Unstructured (non-composite) processes are the basic building blocks**
  Unstructured processes define behaviour which does not require further decomposition. Such processes can be written in state-oriented or monolithic style (but the external synchronisation conventions must be followed).

2.1.2 Data Type Definitions

The data type definitions in an object-based LOTOS specification have two main functions. Firstly, they are used in representing the state of a process. Secondly, they are used with the synchronisation mechanism to establish or negotiate values between processes. The object-based approach suggests using process definitions to represent structural and communication attributes of a system, whilst the data types play a more passive, though equally important, role.

The case study in this paper showed that there is often a more complex inter-relationship between the structure of the data typing and process algebra parts of the specification. The structure of the data provides the basis upon which the system is understood. This structure is then imposed on the process algebra. For example, the network data is a record of the data for each node in the network. Therefore, a network process is composed of a number of node processes. Similarly, the data for each node is divided into routing information (for route calculation) and network information (for communication). This is reflected in the process algebra — a node process is composed of a node-body (for routing) and a node-interface (for communication control).

A fundamental part of writing an object-based LOTOS specification is the matching of structure in the data typing and process algebra. Although the process algebra plays the main structural role in the design, it should be clear that this structure is based on the data. It may be possible to use the ACT-ONE to capture requirements in the initial stages of analysis. Then, as the first stage of design, this statement of requirements can be represented more constructively in full LOTOS. This is beyond the scope of this paper.

2.2 Object-Based Development

The purpose of this paper is to show the effectiveness of LOTOS in representing object-based designs. It is not an examination of a comprehensive object-based design method, but it does promote the notion that such a strategy can be developed. The case study re-used the structure of the analysis as an initial design for a solution. The object-based LOTOS specification is open to formal manipulation, as part of the design process, but this was not investigated.

The case study illustrates how an object-based approach narrows the gap between analysis and design. Consistency of representation, and conceptual congruence between the way in which the problem is defined and the way in which it is solved in an implementation, led us to believe that it was valid to use the structure of the analysis as a design. Much of the work done in analysing a problem is therefore incorporated in the design of a solution. Our initial analysis identified components of the system, i.e a decompositional approach was applied to the whole problem. Each of the components was further decomposed until decomposition was no longer necessary to improve understanding. This implied a purely top-down method, whereas the actual development combined this with a bottom-up strategy. Components, i.e. the object-based processes, were specified such that different parts of the system were at different levels of abstraction during the development of the specification.

The C++ implementation structure was based on the structure of the LOTOS specification. Initially, all LOTOS process definitions had corresponding C++ class definitions. But, for reasons discussed in section 4.3, it was not always reasonable for this constraint to be enforced. The object-based specification removed the need for design decisions in the code. The only implementation concerns, which may be considered structural, involved the removal of non-determinism and concurrency. In effect, the object-based specification defined the requirements of the system and provided a structure on which they could be implemented.

3 Lessons learned

3.1 The Importance of Structure

Much of the emphasis of this work has been on structure. Structure at the specification, design and implementation stages of the software life cycle have been considered. The importance of structure in the specification needs to be examined in the context of object-based development.

There is a definite structure contained in the LOTOS specification of the network routing model. We need to address the question of why it is there. In large, complex specifications it is impracticable to reason about the behaviour of the system as a whole. This suggests that a conceptual decomposition of the problem must be inherent in the specification. We will refer to a specification as being structured only if the decomposition of the problem is explicit.

The main benefit of having a structured specification is that it can be used to aid understanding[4]. Clearly a specification that is easier to understand is also easier to implement. The introduction of structure should also make the system more susceptible to sensible change — the structure should help to control change by encouraging meaningful extensions or alterations and discouraging other fixes.

Arguments against structured specifications concentrate on their constraining nature. Implementers may argue that the decomposition of the system, as captured by the specification, is not the way in which they would choose to structure the code. There may be a conflict of interest between writing the specification to aid understanding and writing it in a way that eases the step towards implementation. This is certainly a concern when implementers are working to specific constraints imposed by the programming

[4]It is likely that a poor structure could also hamper understanding.

language or performance demands. In some cases the conflict of interest between the specification and the implementation may mean that the structure within the specification cannot be followed by the implementers. It is effective use of systems analysis and bottom-up knowledge that mitigates against this.

Within the object-based paradigm, this problem is not as prominent. The decomposition of a system into a set of communicating objects is consistent at all stages of the development cycle. In particular, there is a closer binding between the specification and implementation architectures. The price for this is that object-based specifications are less abstract. The structure of the problem has not been abstracted; instead it has been carefully represented in the specification with the intention that it be used in the implementation.

A perfect scenario would occur when the initial system analysis produced an informal problem decomposition that is acceptable to both specifiers and implementers. Here we mean acceptable in the sense that the decomposition can be used directly by both. In the object-based paradigm this would require the objects within the specification to have counterparts in the implementation. The structure of the system would then be completed by realising the communication between these components.

There are other advantages in reusing the specification structure in the implementation:

- **Generality**
  Writing a structured specification requires making a number of design decisions. In an object-based specification these decisions are often concerned with producing components that are general. This generality can then be exploited for component extension and reuse.

- **Testing**
  Making the specification and implementation structures as isomorphic as possible makes traceability easier, in the sense of a design audit. Testing the system can be done in a bottom-up fashion. This gives more confidence in the code being a valid implementation of the specification.

- **Controlling Change**
  Extending or changing the system can be achieved in a more controlled manner. In an object-based specification, modifications can be kept localised. However, if the implementation has a different structure from the specification then updating the code to match the specification

may require changes across the whole system. Structural compatibility means that changes to the specification can be more easily incorporated in the implementation. Verification of the new system can concentrate on the components that have been altered.

3.2 Suitability of LOTOS for Object-Based Specification

The overall result of the project was that LOTOS is a suitable specification language for capturing the behaviour of a complex system as a collection of interacting concurrent objects. There was an informal correspondence between objects and processes, and between message passing and event synchronisation. This helped to incorporate the structure of the specification in the implementation. A consequence of this was an obvious relationship between the different stages of the development process. The LOTOS specification acts as a formal design. It not only specifies the requirements of the system, but it also provides a framework within which the implementation can be built.

The advantage of a consistent specification style is the ability to structure specifications in such a way that design approach can be explicitly stated. Complex architectures, in particular, require a consistent structured approach to aid comprehension. The object-based LOTOS style seems ideal because of the way it allows for the modelling of systems as interacting parts, each of which can have a straightforward mapping onto real world implementation entities.

3.3 The Question of Inheritance

Work is currently being carried out on introducing the concept of inheritance to LOTOS [1,4]. Our case study did not explore the use of inheritance in the specification, and consequently inheritance was not used in the C++ implementation.

Before inheritance can be used at all stages of the development process, it is important not only that inheritance relationships exist but also that they can be exploited. Libraries of components need to be created and inheritance needs to be used as naturally as any other language construct.

4 The Network Routing Case Study

4.1 An Overview

The objective of the case study was to apply the object-based approach to the development of an implementation of a network routing [5,10,12] algorithm. This involved producing an object-based LOTOS specification of such a system which not only provided a formal statement of the requirements but also acted as a design for a potential solution. A C++ implementation was then coded from this specification. The object-based strategy was to make the step between specification and implementation as simple as possible by retaining the same structure throughout the whole development process. In other words, the design was structured according to problem domain understanding. This design can then be transformed to cope with implementation concerns.

The second part of the case study investigated the object-based claims for extensibility and re-usability. The behaviour of the system was made progressively more complex by extending the routing algorithm. Furthermore, a flow control mechanism was added without changing the existing structure. In twelve elapsed weeks, four LOTOS specifications were written and three of them were implemented. A more complete review of the case study [6] is available on request from the author (together with listings of the LOTOS and C++ text).

4.2 The LOTOS Specifications

The LOTOS specifications produced deal with a dynamic[5] system of nodes connected by links, together with a routing mechanism in each node for the transfer of data between nodes. The network process is used to guarantee the allocation of unique identifications when nodes and links are added to the system. It 'spawns' node processes as they are added to the system. It also synchronises the addition of links (together with the sending of messages down these links) in all current nodes. An informal LOTOS-like specification of this follows:

```
PROCESS Network [add_node,add_link,send,recieve](..):noexit:=
( hide message in
```

[5]New links and nodes could be added at any time during the lifetime of the system.

```
   (add_node; Network [..](..) |[add_link, message]|
               Node [message, add_link, send, receive] (ID, No_Links)
   )[]
   (add_link? node1:Node_Sort ? node2:Node_Sort; Network [..](..)
   )
    []
   ( message? ..;Network[..](..) (* Internal transfer of a message *)
   )
) endproc
```

4.2.1 The Node

Each node in the network corresponds to an instantiation of an infinitely recursive process. Since all the nodes have to be capable of being connected (by links), a single LOTOS gate (**message**) was chosen for all communication between nodes. This requires that all the node processes synchronise at this gate for every communication that occurs. Every node which is not at either end of the link carrying the message (in the network routing model) must participate in the communication, but ignore it. There are only two active participants in the transfer of a message down a link. The addition of links is very simple: every node has a link set which is updated every time a link (which connects it to another node) is added to the network.

The object-based approach requires that each node is always capable of participating in a message event. In the specification, this is achieved by giving each node a node-interface component which always participates in these events, but only routes those which are applicable to that node (by passing them on to the node body). Informally,

```
Node = Node_Interface |[..]| Node_Body
```

4.2.2 The Node-Body

The **Node** process is structured as two processes in parallel: the **Node-Interface** (for controlling synchronisation) and the **Node-Body** (to provide routing behaviour). The **Node-Body** has three components: the buffer for incoming messages, the buffer for outgoing messages, and the router between these buffers. It provides an interface to the environment to allow the addition of new messages to (**send**), and removal of old messages from (**receive**), outside the network. Also, it must communicate with the **Node-Interface** to accept messages for routing from inside the network.

Internal communication between components is straightforward. The **Node-Interface** transfers relevant messages to the **Node-Body** through an **acknowledged** message. The **Router** extracts messages from the **Buffer-In**, routes them by updating one of the message fields, and transfers them to the **Buffer-Out**. These are then sent to the **Node-Interface** via an **out-message**, and finally back onto the network. The **add-link** message results in two components of the **Node** changing their internal state: the **Node-Interface** needs to update its communication data, and the **Router** needs to update its routing data. This is represented in figure 1:

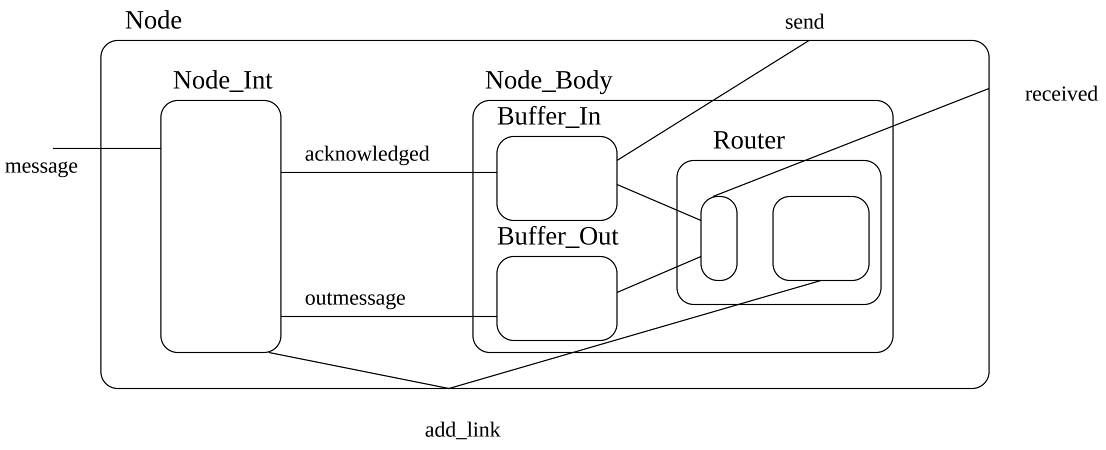

Figure 1: Node Structure Diagram

This structure is replicated in the object-based LOTOS specification as follows:

```
PROCESS NODE_BODY [acknowledged, outmessage, send, receive, add_link]
                  (Node_ID: Node_Sort): noexit:=
(hide next_mess,routed,buff_status in
 BUFFER_IN[acknowledged,next_mess,send](Node_ID, No_Messages)
 |[ next_mess ]|
 ROUTER[next_mess,routed,add_link,receive,buff_status]
       (New_Routing_Data(Node_ID))
 |[ routed,buff_status ]|
 BUFFER_OUT[routed,outmessage,buff_status]
)endproc
where
PROCESS BUFFER_IN [acknowledged,next_mess,send]
        (Node_ID:Node_Sort,messages_in:Messages_Q_Sort):noexit:=
(acknowledged?the_message:Message_Sort;
 BUFFER_IN[..](Node_ID,add_message(the_message,messages_in)) )[]
(next_mess!next(messages_in)[not(empty(messages_in))];
 BUFFER_IN[..](Node_ID,remove_message(messages_in))  )[]
(send!Node_ID ?to:Node_Sort? data:DecDigit;
 BUFFER_IN[..] (add_message
                (Create_Message(Node_ID,to,Create_Routing_Data(L0),data),
                  messages_in))
)endproc

PROCESS BUFFER_OUT[routed,outmessage,buff_status]
                  (messages_out: Message_Q_Sort):noexit:=
(routed? the_message: Message_Sort;
 BUFFER_OUT[..](add_message(the_message,messages_out))
)[]
(outmessage! next(messages_out)[not(empty(messages_out))];
  BUFFER_OUT[..](remove_message(messages_out))
)[]
(buff_status! message_out;
 BUFFER_OUT[..](messages_out)
)endproc

PROCESS ROUTER[next_mess,routed,add_link,receive,buff_status]
              (R_T: Routing_Sort):noexit:=
(* This process is defined as a composition of a table component and
   a routing mechanism component. The complex routing functionality
   is ‘hidden’ in the communication between these. *)
(hide table_access, table_update in
 TABLE[add_link,table_access,table_update](R_T)
 |[ table_access, table_update ]|
 ROUTING[next_mess,routed,table_access,table_update,receive,
         buff_status](ID_Table(R_T)) )endproc
```

This fits in with the object-based strategy as follows. The gates for receiving messages are **acknowledged** (for messages arriving from the network), **send** (for accepting new messages from the environment), and **add-link** (for updating the routing information in the router process). The gates for sending messages are **outmessage** (for sending out messages that have been routed by the node to one of its links), and **receive** (for removing a message from the network that has arrived at its correct destination node). The internal gates are for communication between the component processes.

4.3 Implementing the object-based specification

4.3.1 Understanding the Object-Based LOTOS

From our experience in the network routing case study, at the end of twelve weeks the programmers involved (who were not LOTOS experts) could understand LOTOS specifications provided they were written in a consistent and familiar style, and provided there was expert advice available to make clear any difficult points. The object-based approach helped because there is consistency between the concepts in the programming environment and the style of the specification.

4.3.2 Implementation Decisions

The C++ implementation had to resolve the following aspects of the LOTOS specification.

Concurrency: The network specification was primarily constructed from a number of concurrent node processes. It was not possible (within the scope of the project) to accommodate this concurrency in the C++ implementation. Instead, it was decided to model the concurrency by allowing an arbitrary interleaving of the processing within each of the nodes. This was achieved by requiring, within the implementation, external triggering of the routing and transfer of messages between nodes.

Non-determinism: Internal events in the specification also had to be modelled by a similar mechanism as above. For example, after a node has routed a message it is then ready to route another. The next message to be routed has to be extracted from the message buffer before it can be processed. This is represented in LOTOS by an internal event.

The C++ implementation requires an external triggering of the transfer from buffer to router. This is related to the concurrency issue: in a concurrent implementation the node process could continually route messages without interruption.

Multi-way Synchronisation: In the specification, event synchronisation was used for the transfer of messages between nodes, causing two problems. Firstly, all the nodes in the specification have to synchronise on the message event even though it is applicable to just two of them. Secondly, event synchronisation is used to transfer data on the link in either direction. In the implementation, links were coded as pointers to nodes. This means that a direction is implied for each message, and two pointers are needed for each link. The external triggering of messages being sent down links removes the problems created by the multi-process synchronisation in the specification. This implementation decision means that there is no need for a node interface component in the C++ code.

4.3.3 Re-using Structure

There was a conscious decision to replicate the design decisions made at the specification stage during coding. For example, the decomposition of a particular process into a system of processes was reflected in the implementation by splitting an object into a corresponding system of objects. (It should be noted that structure need not be enforced at the lower levels of coding. The differences between the two languages means that it may be possible to directly implement, i.e. without further decomposition, a process that was defined as a composition of other processes.)

The C++ header file[6] of a node class is given below.

```
class node
{private:BufferLink* outBuffers;
         Buffer* inBuffer;
         char* nodeName;
         Buffer* routeMessage(Message* mesg);
 public: node(char* name);  \\ instantiation function
         void addlink(node* newNode, char* linkId);
```

[6]A C++ header file gives an external view of the behaviour of a class without including internal details.

```
void messagein(Message* mesg);
void send(char* MessageID,char* dest,char* data);
Message* receive (char* linkId);
void process(); \\ external triggering of the router
node* message_out(char* linkId);
```

The structure of the node specification was used in the design of the node implementation. However, the implementation decisions (outlined above) meant that there was no need for a node interface. Furthermore, the message passing had to be 'divided in two' so that the C++ node could distinguish between incoming and outgoing messages. This division resulted in two separate message links going directly to the **in-links** and **out-links** components. In all other respects, the structure of the implementation corresponds to the structure of the specification.

4.4 The Iterative Development Approach

Our initial aim of developing the implementation from the specification was fulfilled. Design (structure) decisions were made after consultation between specifiers and implementers. Often, whilst implementing some part of the system, a more meaningful decomposition of the problem was discovered. This was recognised by making alterations to the specification. For example, the separation of the routing process from the routing data occurred half way through the development of the specificaton — it made the system much more amenable to extension. In a general sense, the specification design was developed top-down — processes were split into sub-processes and these sub-processes were then further split until each component process was simple enough not to require further decomposition. Sometimes problems occurred in the sub-processes that required some of the previous design decisions to be changed. The development would then continue as before. The conventional view of the object-based approach is that it is compositional (bottom-up). In a sense this is true[7], but there has to be an initial decomposition of the problem before base components can be identified.

System development is then bottom-up. Simple components are specified and implemented together. These can then be combined with other components to produce more complex components, which are also tested. The

[7]Perhaps a compositional approach should not mean that code is written bottom-up, but that it can be understood bottom-up.

complete system is constructed in this way. There is flexibility within this development method. The decomposition of the system can be changed by altering the balance of functionality between the sub-components of each component. This encourages the development team to work to a definite structure, but does not discourage change when it is needed.

The object-based approach is both compositional and decompositional, providing a degree of flexibility to decide on the balance between the two methods. Traditionally (in the object-based community) the balance has been in favour of the bottom-up method since this provides the programmers with an effective rapid prototyping mechanism.

4.5 Testing

One of the main advantages of using a structured specification is in the area of testing. If the structure of the specification is followed in the implementation, verification can be undertaken not only at the system level but also at the component level. This is beneficial to the development because verification can begin at an earlier stage and at a finer level of granularity. It should be noted that, by writing stubs for components with limited functionality, incremental testing was also done using a top-down approach. This method is more difficult to apply extensively but it was used initially to test network behaviour in the absence of routing algorithms.

Testing also increased understanding of the problem domain. For example, we discovered that some routing algorithms depend on bi-directional links (for example, the *Hot Potato* algorithm). This was not clear from the informal analysis of the problem.

4.6 Extensions to the Behaviour

The behaviour of the system was made progressively more complex by extending the routing mechanisms. Flexibility was built into the design to allow for extensions or alterations to the routing process. Checking extensions concentrated on testing the new routing mechanisms, because the other parts of the system had already been thoroughly tested when the first mechanism was developed. This was possible because of the way in which changes to the system affected only the components which were involved in the routing process. Localised changes like this are made possible when work is done to reduce the coupling between the components in the system.

Our extensions were restricted to changing some of the components in the original system. This illustrated that specification and code re-use was possible, but it also showed that it is not always done in the most elegant or efficient way. The lessons that we learned from carrying out the extensions were:

- Meaningful generalisation should be applied as much as possible: the extra time needed in producing a flexible structure results in benefits later on.
- The object-based approach makes it easier to change behaviour by replacing compatible components.
- Components that can be extended (within one application area) are not the same as reusable components. For example, the routing mechanism is replaceable but it is unlikely that it could be used in the specification of different applications.

5 Conclusion

We showed that working from a formal specification has many advantages. It enhances the understanding of the problem, and consequently removes potential errors earlier on in the development cycle. It also provides a formal contract between specifiers and implementers. Making design decisions explicit rather than having them hidden (as often happens with an informal specification) is beneficial. Also, removing non-determinism makes implementers more aware of the choices that they are making when they write the code.

The advantages of following the object-based paradigm are as follows. The conceptual integrity between all stages of the development cycle allows a consistency of representation that makes it easier for all members of the development team to reason about the system together. Generalisation and specialisation in the specification can be reflected in the implementation to make components easier to reuse. Extensions to the system can quite easily be carried through from the specification (thus controlling how the system evolves). Also, by using the combined top-down/bottom-up approach, the development of the specification and implementation can take place in parallel. This is more realistic than a scenario in which an implementation team is given a completed specification.

Structure was shown to be an important part of the design stage. It aids understanding, and structure in the specification can, and should wherever possible, be reused by the implementation (provided the specification models objects in the real world).

This paper is based upon a small investigative case study. More work needs to be done on:

- introducing the concepts of inheritance to the object-based work.
- applying this approach to a much larger project where the real benefit of the object-based approach can be fully investigated.
- examining the exact nature of the relationship between the specification(s) and implementation(s) produced in this way.

It has been shown that there is potential for applying this development approach to other (more complex) systems. LOTOS seems an ideal vehicle for producing designs of object-based distributed systems.

Acknowledgements

This work was funded by the British Telecom Group Technology and Development Services project on Formal Methods Application. It was carried out as part of the author's PhD research at Stirling University with financial support from the Department of Education for Northern Ireland (DENI).

The author acknowledges the contribution made by David Freer to the case study. He was the primary C++ coder and also helped in system design. Thanks also to the BT Formal Methods Unit (Ipswich): Steve Rudkin, Steve Colwill, Jeremy Wilson, Rob Booth, Jim Lynch and Elspeth Cusack. Special mention goes to Ken Turner, Bob Clark and Ashley McClenaghan in Stirling for their support.

References

- [1]: S. Black, *Objects and LOTOS*, Draft Technical Report, Hewlett-Packard Laboratories, Stoke Gifford, Bristol, October 1989.

- [2]: R.G. Clark, *Using LOTOS in the Object-Based Development of Embedded Systems*, Proceedings of the IMA Conference on the Unified Computation Laboratory, Stirling University, July 1990.
- [3]: B.J. Cox, *Object Oriented Programming — an Evolutionary Approach*, Addison-Wesley, 1987.
- [4]: E. Cusack, S. Rudkin, C. Smith, *An Object Oriented Interpretation of LOTOS*, The 2nd International Conference on Formal Description Techniques (FORTE 89), December 1989.
- [5]: D. L. Davies, D. L. A. Barber, *Computer Networks and their Protocols*, Section 3, John Wiley, 1979.
- [6]: P. Gibson, D. Freer, *Applying LOTOS in an Object Oriented Development Strategy: an investigative case study*, BTRL, Formal Methods Group, Internal Report, 1991.
- [7]: B. Liskov, J. Guttag, *Abstraction and Specification in Program Development*, MIT Press 1986.
- [8]: LOTOS IS 8807, *LOTOS — a Formal Description Technique based on the Temporal Ordering of Observed Behaviour*, 1988.
- [9]: Bertrand Meyer, *Object Oriented Software Construction*, Prentice-Hall International Series in Computing Science, 1988.
- [10]: M. Schwartz, T. E. Stern, *Routing Techniques used in Communication Networks*, IEEE Transactions on Communications, VOL COM. 28, No. 4, April 1980.
- [11]: Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.
- [12]: A. S. Tanebaum, *Computer Networks*, Section 5.2 (pp 196 – 214), Prentice-Hall, 1981.
- [13]: K. J. Turner, *Formal Description Techniques II*, (pp 117 — 133), *A LOTOS Based Development Strategy*, December 1989.

- [14]: C.A. Vissers, G. Scollo, M. van Sinderen and E. Brinksma, *On the Use of Specification Styles in the Design of Distributed Systems*, University of Twente, Fac. Informatics, 7500 AE Enschede, NL, 1989.