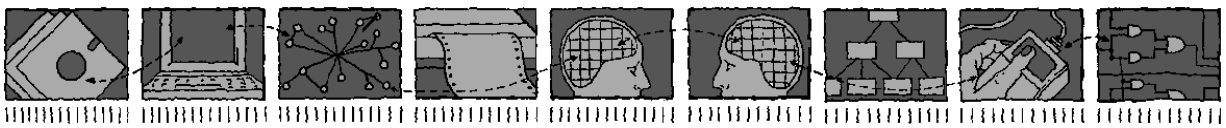
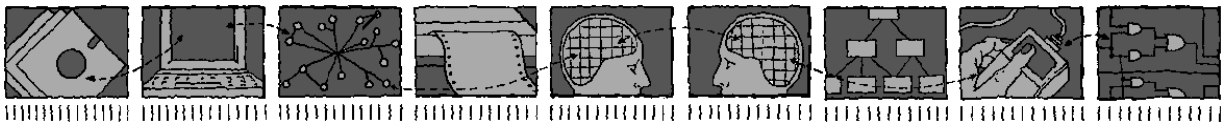


Department of Computing Science and Mathematics
University of Stirling



An Architecture Based Approach to Specifying Distributed Systems in LOTOS and Z

Richard O. Sinnott

Technical Report CSM-144

October 1997

Department of Computing Science and Mathematics
University of Stirling

An Architecture Based Approach to Specifying Distributed Systems in LOTOS and Z

Richard O. Sinnott

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

Telephone +44-786-467421, Facsimile +44-786-464551
Email sinnott@fokus.gmd.de

Technical Report CSM-144

October 1997

Abstract

Specification is difficult. It is often the case that the most difficult aspect of specifying is the structuring of the specification to begin with. Adopting an architectural approach can help to alleviate this structuring problem. We have investigated how the formal languages LOTOS and Z can be used to develop specification templates suitable for architecting specifications of distributed systems.

The immediate question that arises is: where do the architectural concepts come from? We have focused primarily on the work of the current standardisation activity of Open Distributed Processing (ODP). The approach taken there is to provide an object-oriented set of concepts and to use these as the basis for developing a multi-viewpoint approach. A viewpoint may be regarded as an abstraction of the system focusing on some particular aspect, the intention being to help reduce the complexity of the system as a whole. ODP identifies five viewpoints: the enterprise, information, computational, engineering and technology viewpoints. In our work, we formalise the foundation set of concepts in LOTOS and Z then show how specification architectures based on the computational viewpoint can be developed. We also highlight the advantages in a formal approach through the identification of limitations and errors in the ODP framework.

Central to work on distributed systems (or any system adopting an object-oriented methodology), and to the computational viewpoint of ODP in particular, is the issue of type management. We have investigated how LOTOS and Z can be used to reason about type systems generally and to investigate the issues that have to be resolved in determining type equivalence. We base our work on the idea of type equivalence as a substitutability relation between types. We address issues such as signature type checking, behavioural type checking and non-functional aspects of type checking such as quality of service. Our investigations have also included a treatment of multimedia type systems where continuous flows of information raise performance issues that are particularly important.

We have applied our approach to two case studies: the specification of the ODP trader in LOTOS and the specification of a producer and consumer flow configuration in Z. The advantages and disadvantages of adopting LOTOS or Z to develop specification architectures for distributed systems are highlighted and discussed.

Contents

List of Figures

Chapter 1

Introduction

In this chapter we provide the context in which this thesis was developed and give an overview of the work as a whole. We begin with an overview of distributed systems and their need for formal representation. The role of object-orientation and its relevance to distributed systems is discussed. Following this we hypothesise that an architectural approach can reduce the complexity in specifying distributed systems. In particular we propose that the formal languages LOTOS [101] and Z [106] can be used to develop specification architectures suitable for constructing specifications of distributed systems. This hypothesis forms the basis for the investigations documented in this thesis.

1.1 Context of Thesis

This thesis is concerned with writing specifications of distributed systems. Writing specifications is a difficult activity. Writing specifications of distributed systems is especially complex due to their inherent properties, for example, their potential remoteness and heterogeneity. We provide a more detailed discussion of these properties in section 2.1.1. It is quite possible to write specifications for a particular problem domain without recourse to any reference architecture developed for that domain. We argue that more insight into formal languages and the domain itself is generated when they are applied to the architectural principles that underlie that domain.

We have investigated how the formal languages LOTOS [101] and Z [106, 181] can be used to develop specification architectures suitable for constructing specifications of distributed systems. These formal techniques are particularly interesting since they may be seen in many respects as coming from opposite ends of the formal methods spectrum. An overview of this spectrum and the positioning of LOTOS and Z in the family of specification languages is given in chapter 3.

1.1.1 Open Distributed Systems

Open distributed systems are one of the most important developments in computing science today [198, 66, 107]. The concepts and technologies that underlie them are revolutionising computing science. The typical large mainframe with collections of terminals jostling for resources is rapidly becoming obsolete. The primary reason for this is the decreasing cost of computer hardware and increased power of modern computers. Open distributed systems have numerous advantages (and disadvantages) associated with them. For example, distributed systems can be designed so that they are more tolerant of certain system failures and make better usage of system resources. On the other hand, distributed systems are, in the main, more complex than other computer systems. These advantages and disadvantages are discussed in more detail in section 2.1.2.

Distributed systems themselves consist in the main of collections of computing systems, where a degenerate computing system is a single computer. These systems communicate with one another typically using a sophisticated network. There is a fundamental distinction between a distributed system and a basic network of computers however. Loosely, the distinction is the ability of distributed systems to mask

complexity from the user. That is, through providing specific features (*transparencies*) that allow aspects of the inherent distribution to be hidden. For example, the remoteness of applications and usage of potentially heterogeneous resources can be hidden from users of the system. We discuss these and numerous other complexities that are associated with distributed systems in section 2.1.3 along with features they are expected to provide for masking these complexities. We note here that object technology offers a way to manage certain of these complexities.

The Openness of Systems

The term *open* is frequently found in literature relating to distributed systems. There are numerous interpretations as to what exactly this term means in the context of distributed systems. This ambiguity is captured by Leopold et al. [129] They identify several definitions that can be considered as correctly interpreting the term "open" as found in distributed systems literature:

- an environment and its model are easily obtainable and well documented;
- the model and environment exist on many operating systems and work with many programming methodologies and languages;
- several manufacturers support and control the market;
- the environment and its model have developed after considerable "open debating".

We regard the openness of a distributed system in this thesis as the extendability of that system. That is, how new resource-sharing systems can be added to a system without disrupting existing services. Typically this is achieved through making available the descriptions of specific entry points into the system, commonly referred to as interfaces. From these, it should be possible to add new (possibly heterogeneous) hardware and software to the system. This definition may be regarded as being more abstract than those given in [129]. That is, we do not concern ourselves with the specifics of proprietary hardware and software and their interworking in this thesis, nor what constitutes a resource-sharing system. Rather, since our interest is in developing formal specifications of such systems, we abstract from such issues. The usefulness of abstraction in a formal methods context is discussed in more detail in section 3.2.

As well as this task, recent areas of research [4, 107] into open distributed systems are replacing the notion of *interconnectivity* of computers by *interworking* of enterprises. We note the distinction between these two concepts here as they pertain to this thesis. We regard interconnectivity of computers as the ability of computers to communicate successfully with one another. This might be realised, for example, through ensuring that they use similar communication protocols as might be found in the Open System Interconnection (OSI) reference model [103]. Interworking is wider in scope than message passing capabilities though. That is, whilst communication is essential for the successful interworking of separate enterprises, it is only a basis on which the interworking can be established. Interworking may thus be regarded informally as the integration of enterprises to achieve some commonly agreed goal. This integration need not be complete between enterprises. That is, separate sub-enterprises of some larger enterprise may be integrated to achieve some task. An overview of the relationship between interconnectivity as might be found in the OSI domain to interworking as might be found in ODP is given in [129].

In this thesis we adopt one architecture developed to enable the issues involved in interworking to be addressed: the current standardisation activity of Open Distributed Processing (ODP) [107, 108, 109, 110, 111]. It is clear that research into enterprise interworking is inherently multifaceted. Techniques for alleviating this complexity are thus highly desirable. To this end ODP has embraced object-oriented technology and formal techniques.

1.1.2 Object Technology and Distributed Systems

Object technology is regarded by many to be the panacea for software development [27, 46, 137, 165]. To others it is simply an over-hyped computer trend [67]. Whatever point of view is taken, object technology is certainly in widespread use throughout the computing industry.

Object technology pervades all aspects of computing science: system analysis and design, programming etc. Despite the obvious popularity of the object-oriented paradigm, a phenomenal amount of disagreement exists on the fundamental terminology. Gibson [81] captures this through lists of different definitions for popular object-oriented concepts, all of which, he argues, are correct in a given context. To some extent these terminological irregularities of fundamental definitions stem from the popularity of object technology. For example, an object to a system designer is likely to be different to that of a programmer. As captured by Gamma et al. [77],

"everything down to the hardware or all the way up to entire design applications can be an object. How can we decide what should be an object? ".

An alternative perspective on the lack of precision in existing object-oriented technology is given by Danforth and Tomlinson [57] who state that:

"*object-oriented programming is generally expressed in philosophical terms, resulting in a natural proliferation of opinions concerning exactly what object-oriented programming is*".

The essence of object-orientation is that concepts, ideas, processes or data, or combinations of these can be grouped together into a capsule called an object. An object has associated with it at least one interface[1], which is used to communicate with other objects. The only way for one object to access information associated with another remote object is through interacting with that object at one of its interfaces. This hiding of information is commonly referred to as encapsulation [27, 46, 165].

Thus object-orientation offers several immediate advantages with regard to distributed systems [18, 152]. Encapsulation enables information to be hidden. Interfaces can be used to allow restricted access to areas and hence information in the system. This compartmentalising of the system information provides a more structured approach to system design and management. In particular, the approach lends itself to openness as defined above.

Object-orientation and formal techniques, it has been argued [120, 147], are in many senses complementary to one another. The proliferation of object-oriented specification techniques [14, 81, 118] is testimony to this. An overview of object-oriented specification approaches is given in [123] and [186]. Similarly, numerous approaches to introduce object-oriented principles and practices into non-object-oriented formal development techniques are occurring, [147] being one such example.

1.1.3 Formal Techniques for Describing Distributed Systems

Formal techniques in various guises have been around for a considerable amount of time. Surprisingly, little work has been done to establish which language is best suited to particular problem domains. More often than not their usage is predominantly given by existing expertise in that technique, rather than their being more suited for the task at hand. Of course, certain languages were developed with certain problem domains in mind. For example, Estelle [100], LOTOS [101] and SDL [118] were developed with the intention of being suited for specifying communication protocols. Hence it is natural that they are more suited to this problem domain than other techniques. Other techniques such as Z [106], it could be argued, were developed more with the idea of software engineering in general. That is, as an aid to developing code in a more formal manner. The issue that we are interested in is, what happens when a new problem domain arises that existing techniques were not specifically developed for? Needless to say, the developers of the formal languages would like to apply their language to new problem domains. As stated by [195], this is especially true with the standardised formal techniques LOTOS, Estelle, SDL (and soon to be standardised Z), where the investment in their development has been considerable.

Distributed systems represent one such untried problem domain offering a new and interesting challenge for formal techniques. At present it is not clear which technique is best suited for specifying and reasoning about distributed systems. There are several possibilities in determining which formal technique is more

[1] The number of interfaces an object has is discussed in section 2.3.2.

suited. The low level approach would be to try to specify particular (aspects of) distributed systems and learn from the experience. This is not a particularly satisfactory solution since it may result in many tried and failed specifications in many different languages. We argue that a higher level and more useful approach is to develop specification architectures.

The term architecture is receiving more attention from the computing science community [77, 107, 195, 196]. As inheritance offers the possibility of code re-use, so architectures offer the re-use of designs and experiences. To create an architecture requires detailed knowledge of the underlying problem domain. From this understanding, precise concepts and structuring rules for those concepts, as well as a basic terminology itself for reasoning about that problem domain can be developed. Such a collection of concepts, rules and terminology for a particular problem domain is common termed a *reference architecture*.

We argue that, given a particular domain (here distributed systems) and an associated domain specific reference architecture, the ability to formalise the concepts and structuring rules associated with that architecture (in a given formal language) may be regarded as a better yardstick for language comparison in that domain. Thus if a given formal language may be used to formalise all aspects of a given reference architecture then the ability to build specifications based on that architecture should be straightforward in that formal language. Alternatively, if a formal language cannot be used to formalise the architectural concepts or structuring rules associated with those concepts, then its applicability for specifying systems in that problem domain needs to be questioned.

The immediate question that arises is where do these architectures come from? In this thesis we are concerned with the specification of distributed systems. Hence it is natural that we should have an architecture that lends itself to reasoning about distributed systems. There exist several currently proposed architectures for developing distributed systems. [66, 198] discusses the relative merits of some of these proposals through worked examples. In this thesis we have adopted one particular architecture: the Reference Model of Open Distributed Processing (ODP-RM[2]) [107, 108, 109, 110, 111]. This may be regarded as the most wide-reaching of the proposed architectures. That is, whilst certain architectures attempt to provide open interconnectivity as discussed earlier in section 1.1.1, ODP has attempted to address entire enterprise interworking. We discuss ODP in relation to these other architectures in section 2.3.

1.2 Contributions of the Thesis

The main contribution of this thesis is the investigation of how LOTOS and Z can be applied to specify architectures of distributed systems. And in turn, how specifications of distributed systems can be constructed. In doing this, we contribute to the body of knowledge on formal techniques, object-orientation and distributed systems.

To achieve such specification architectures, we consider how the basic (architectural) concepts that underlie distributed systems might be formalised in LOTOS and Z. Further, we consider how these more elementary concepts can be structured (composed) with one another. From these investigations we conclude that neither formal technique has the necessary features in its classical usage to model distributed architectures completely. As a result, we propose a novel approach to specifying (aspects of) open distributed systems. This new approach enables us to specify exactly what is required when composing two (or more) systems together to enable behavioural interworking to take place. The novelty here is that we can specify inside a specification the necessary criteria for interworking to take place. That is, it is normally the case that reasoning about a specification takes place outside the specification, *e.g.* when testing for the satisfaction of some particular behavioural relation. We have developed an approach whereby these testing relations can be incorporated into the specification itself. Further, we extend the basic notion of behavioural relations as found in languages such as LOTOS, *e.g.* conformance, extension and reduction [35, 125], to include performance and other non-functional aspects related to the behaviour that can affect the successful interworking of enterprises.

Traditionally systems could be composed provided they were syntactically compatible. That is, they could understand the messages that were sent between them, i.e. they were syntactically type equivalent.

[2] It should be noted that this is the correct acronym for the ODP reference model and not RM-ODP.

Type equality based on syntactic structuring only is myopic if interworking as opposed to interconnectivity is to be achieved, though. Syntactic type checking is a well investigated area, though, with a large body of theory and practical approaches to support it. Our approach does not attempt to replace type checking as it currently exists. Rather we augment the syntactic type checking with behavioural checking and other checks that the syntax and behaviour do not capture. Our work may thus be regarded as an extension to the basic "message is understood" paradigm inherent in most type checking systems.

As well as this, our work recognises the particular requirements that distributed systems have on type systems. The client/server paradigm which forms the basis for many distributed systems architectures imposes constraints on type systems that existing approaches do not address. We investigate these differences and show how they can be dealt with and specified from a syntactic, behavioural and non-functional perspective to enable interworking.

Finally, since the architecture we base our work on incorporates (multimedia) flows of information, we investigate how such multimedia flows can be specified and reasoned about. This includes composition of flows and performance aspects related to those flows that might affect successful composition. Thus our approach offers the specifier a powerful means to specify and reason about *open* distributed multimedia systems.

1.3 Related Work

The application of formal techniques and object-orientation is not a new idea. With regard to LOTOS, [14, 44, 52, 147] have all investigated how object-oriented concepts can be represented in the language. Possible extensions to the language have been suggested by [164] for support of the object-oriented concept of inheritance. Gibson [81] considered how the LOTOS data typing language ACT ONE [65] could be used to develop software in a formal object-oriented manner. We note that whilst developing an architectural approach based on concepts derived from object technology, our work does not attempt to provide a new object-oriented development methodology. Rather, we use object-oriented concepts and their formalisation as a tool for reasoning about distributed systems.

There exists a plethora of work related to object-orientation and Z [2, 41, 55, 122, 166]. Overviews of these approaches are presented in [123] and [186]. This work can be decomposed into two main areas: extensions to the language and specification style conventions. The work presented here investigates these two areas and shows how they are limited for our purposes. A new approach is then provided that does not require extensions to the Z language.

Architectural approaches to specifying distributed systems are not a new idea either. [20, 136, 168, 203] have all developed architectural approaches to specify distributed systems. All of these have focused predominantly on LOTOS and not attempted to compare other formal languages in any detail. Vogel's work [203] in particular is relevant to this thesis since he developed an architectural methodology in LOTOS based on ODP. Whilst philosophically similar in approach, our work differs in technical content, his approach being based on a syntactic structuring of specification stakeholders, where a stakeholder may be regarded as a the shell of a process definition that requires behavioural aspects to be inserted.

Gotzhein [86] has also developed an architectural approach to distributed systems development. He presents an abstract mathematical model consisting of objects, actions and interaction points — concepts that we discuss in more detail in chapter 4. Using these basic concepts and some functions that relate them to one another, he shows how more complex structures can be built. He then proceeds to show how his abstract approach can be mapped into a temporal logic specification. Thus he argues that a basis for specification language comparison can be achieved in a formal manner. In a similar vein, Stefani and Najm [148] propose a direct mathematical formalisation of the computational viewpoint of ODP. Their approach and the advantages and disadvantages associated with a direct mathematical interpretation generally are discussed in section 4.1.

Performance aspects and their formal specification have been considered in numerous works. McClenaghan [136] provided extensions to LOTOS to enable performance aspects to be specified. These extensions included timing issues, probabilistic behaviours and prioritised behaviours. Schieferdecker [167] also investigated how LOTOS might be extended to deal with performance oriented aspects. In particular she proposed structured actions. These may be regarded as events in LOTOS parameterised with

time, priority, resource and monitoring signals.

There exists a plethora of time extensions to LOTOS and process algebras generally. An overview of these is given in [153]. A case study for assessing timed formal description techniques is given in [128].

Multimedia distributed systems were considered by Blair [19] using a combination of LOTOS and temporal logic. Issues in the specification of multimedia were covered in the TEMPO project [16]. This work is also being extended [17] to deal with probablistic behaviours. Pinto [159] proposed a new model for the specification of interactive and distributed multimedia systems using a LOTOS-like language.

Surprisingly, very little work has been done using Z for reasoning about distributed systems, architectural approaches or performance aspects generally. Raymond et al. [188] showed how Z might be used to specify aspects of distributed systems. Of particular relevance to this thesis is the work by Fogg et al. [75] who compared LOTOS and Z in their applicability to specify distributed systems. This work suggested, perhaps surprisingly, that when used in the classical style of state and operations in conjunction with a temporal logic, Z was superior to LOTOS for specifying distributed systems. It was argued that LOTOS had two main problems for specifying distributed systems. Firstly, there is the problem of being unable to treat processes as first class entities, *e.g.* pass them around as parameters. Secondly, there is the lack of LOTOS features for reasoning about the specification itself, for example, the inability to describe and reason about traces inside the specification. In Z, this is possible since actions are represented by schemas in Z which are also data types. We discuss this issue in more detail in sections 4.2.2 and 7.5.

Fidge [70] showed how Z could be used to specify and verify real-time behaviour. His approach was based on interpreting Real-Time Logic (RTL) [119] in Z, and subsequently using the features of the Z language for reasoning and verifying real-time behaviours. Such work is rare however in comparison to the abundance of material on timed process algebras.

The use of object-oriented versions of Z has been investigated in some detail for specifying multimedia systems. Indeed the standardisation area of Presentation Environment for Multimedia Objects (PREMO) [117] is actively using a Z extension to reason about multimedia systems. Our work is more general than this in that we look at behavioural issues generally, with multimedia flows being one particular type of behaviour.

Interesting work in relation to this thesis is that of Houston and Josephs [97]. They have been applying Z as an aid to understanding the Object Management Group's (OMG) Core Object Model [90]. We discuss this object model and the work of OMG in general in section 2.3.1.

With regard to formal techniques and ODP there exists a copious amount of recent works, in particular approaches based on the viewpoint languages of ODP [109]. These viewpoint languages and other aspects of ODP are presented in more detail in chapters 4 and 5. Farooqui [68] proposed a LOTOS based approach to ODP systems development based on the idea of viewpoint transformations. His proposal consisted of applying correctness preserving transformations in LOTOS to show consistency between viewpoint languages specifications. This work was limited in that it could only be applied to computational and engineering viewpoint specifications however. Bowman et al. [31] have considered how LOTOS and Z can be used to provide viewpoint consistency. Their work has attempted to embrace inter-language consistency and intra-language consistency across the ODP viewpoint languages. We discuss the relevance of their work to ours in numerous places throughout this thesis.

Stefani [184] and Leydekkers et al. [80] have applied formal techniques to enable reasoning about aspects of quality of service of ODP computational objects. Their work and the wealth of other work related to the specification of ODP systems is discussed in more detail throughout this thesis.

ODP thus represents a major research area that the formal methods community is actively investigating, both in developing new formal languages and new specification styles. Indeed, current standardisation activities [112] are being developed with the intention of being suitable for specifying features of distributed systems. We discuss these desired features in section 3.2 and consider how successfully LOTOS and Z can be used to model them.

1.4 Thesis Structure

The rest of the thesis is structured as follows:

- **Chapter 2 —** gives an overview of distributed systems and their associated properties. The advantages of distributed systems over centralised systems are discussed along with their potential disadvantages. Object orientation is then introduced as a paradigm for reasoning about, modelling (specifying) and subsequently developing distributed systems. In particular we focus on general aspects of object-orientation that aid in the removal of system complexity as well specific object oriented concepts that are especially relevant to distributed systems, namely type and class and the associated concepts of subtype and subclass. The chapter concludes with an overview of current distributed system reference architectures. Our attention focuses in particular on how other approaches compare in scope and technical content to the architecture on which this thesis was based: the ODP-RM.

- **Chapter 3 —** identifies desirable properties that formal methods should possess given the discussion of the previous chapter. LOTOS and Z and their positioning in the family of formal specification languages are presented. Particular emphasis is placed on those specification languages:

  - developed especially for specifying object-oriented systems;
  - that have been used for specifying object-oriented systems;
  - that have been used for specifying and reasoning about performance aspects;
  - that have been used for specifying and reasoning about multimedia flows of information.

- **Chapter 4 —** considers how LOTOS and Z may be applied to develop a generic framework of architectural components based upon Part 2 of the ODP-RM [108]. Some possible approaches to formalisation are presented and their relative merits are discussed. The advantages and disadvantages of both techniques for developing these architectural components are discussed and a novel approach is presented based on these discussions.

- **Chapter 5 —** considers how LOTOS and Z can be used to develop a more prescriptive set of architectural components based on the viewpoint languages of the ODP-RM [109]. In particular we consider the computational viewpoint language. The advantages and disadvantages of LOTOS and Z are discussed and the advantages of the new approach are shown.

- **Chapter 6 —** considers how performance aspects and other non-functional aspects related to distributed systems can be included in the realm of specification. We firstly provide an overview of various performance and non-functional issues that might be needed when considering distributed systems. An overview of current approaches to specifying these aspects is presented and their limitations for our purposes identified. Finally our approach to specifying these non-functional aspects is given and some conclusions are drawn.

- **Chapter 7 —** considers the notion of type and issues relating to establishing type equivalence. Whilst current technologies have adopted a notion of type equivalence based on syntactic aspects of the type, we argue that it is necessary to consider other aspects if open distributed systems are to interwork correctly. More specifically, we state that syntactic type checking should be augmented with behavioural type checking and non-functional aspects of the type, *e.g.* aspects of the type related to quality of service. We provide a specification approach that incorporates all of these aspects in establishing type equivalence.

- **Chapter 8 —** provides specification case studies of how the architectural approach developed can be applied. We show this through the specification of the ODP Trader [115] in LOTOS and a producer and consumer flow configuration in Z.

- **Chapter 9 —** draws some conclusions from the thesis as a whole and identifies possible areas of future work.

- **Appendix A —** shows how the syntactic type checking approach developed for LOTOS can be applied to develop a type management system.

- **Appendix B —** gives a brief summary of the Z notation used in this thesis.
- **Appendix C —** gives a brief summary of the LOTOS notation used in this thesis.

Chapter 2

Distributed Systems and Technologies

This chapter introduces distributed systems in the large and the features inherent to them. An overview of the advantages (and disadvantages) of distributed systems over centralised computing systems is presented.

Following these discussions we provide a broad overview of object technology. In particular we focus on those aspects of object technology that make it especially pertinent to distributed systems. This includes those features of object technology that encourage simplification through abstraction, as well as specific concepts such as type and class. Whilst object technology represents an intuitively appealing paradigm within which distributed systems may be reasoned about and developed, it is not without its drawbacks when applied to distributed systems. These limitations will be discussed in some detail.

Finally we discuss the role of architecture in systems development. The advantages of an architectural approach are discussed, especially those based upon object technology. Current architectural approaches to distributed systems development are then contrasted with one another. We focus in some detail on how other approaches compare with ODP in both scope and content.

This chapter may thus be regarded as setting the context for distributed systems and object technology as used in the rest of this thesis.

2.1 Distributed Systems and Technologies

Distributed systems are reshaping the way in which computing is done today [5]. Their primary goal is to enable applications to interwork and data to be shared between *organisations*. Diversity in hardware and software is a fact of life. This diversity is in some respects due to the explosive growth rate of computing today. Computing science, it could be argued, is a unique area of research. There are few subjects that have such investments associated with them, or such potential for direct exploitation. As a result, the frontiers of computing science are perpetually being extended. Distributed systems represent one area in particular where this dynamism is especially evident.

This rapid growth is not without its problems. As new technologies emerge, they are expected to interwork with older (legacy) systems. These systems are potentially huge, *e.g.* telecommunication systems, often with no central design or management authority. Further, such systems often have stringent requirements attached to them that make incremental changes especially difficult. For example, some systems must operate continuously and system closure to enable repairs or upgrades is often not possible.

Current research into distributed systems and the technologies that underlie them is investigating these problems. These investigations may loosely be placed into two related groups: considerations for the user of the system and considerations for extensions to the system. The first of these corresponds to the need for masking system complexity from the user — commonly referred to as *transparency.* Various forms of transparency have been identified and are discussed in section 2.1.3. The second consideration focuses on the need for openness as discussed in section 1.1.1.

Inevitably, the provision of system support to overcome problems of distribution is difficult. This difficulty stems from the characteristics inherent to distributed systems in the large. We now provide an overview of some of these more frequently occurring characteristics.

2.1.1 Characteristics of Distributed Systems

Typically distributed systems have characteristics such as:

Remoteness: components may be spread over space with both local and remote interactions possible. This in turn introduces a whole host of associated problems that have to be overcome.

Heterogeneity: different technologies may be used within the system — this includes different hardware and software. This too introduces a wealth of issues that have to be resolved. For example, naming issues have to be addressed, *e.g.* translating the names associated with interactions between local and remote components of the system.

Concurrency: components are likely to be executing in parallel. As a result, issues such as synchronisation of components and consistency of shared data have to be addressed. This in turn requires management of separately executing processes and the problems they incur, *e.g.* deadlocks, livelocks, etc. This too is further complicated by the fact that there is often no single controller or scheduler as in most centralised systems. Of particular importance is the need to ensure that the functionality of the system as a whole is represented by the functionality of the concurrently executing components of the system.

Partial Failures: components may fail independently of others. >From this, issues related to error recovery mechanisms have to be addressed.

Asynchrony: global communications are not driven by a global clock. Thus interactions between components may be delayed, which in turn may result in error propagation during communications, *e.g.* when repeated requests are sent to a component that is busy processing an earlier request, thus causing overloading problems.

Autonomy: different parts of the system may be owned and managed separately. This requires a multitude of issues to be addressed. For example, security and protection aspects have to be dealt with. Often this is likely to be more complex than in a centralised system since aspects such as trust and third party interactions have to be resolved. Similarly, resource autonomy means that accounting and charging for resources is not necessarily straightforward. Scheduling and managing the applications that run in the system is also made more difficult since there will often be no single controlling component, as might be found in a centralised system.

Evolution: the technologies in the system may change over time. This requires that systems should allow unwanted components to be removed and replaced with different technologies. It is often the case that older (legacy) systems cannot be replaced, but are expected to interwork with newer technologies.

Mobility: sources and sinks of information in the system might be physically mobile.

As can be seen, these characteristics greatly increase the complexity of distributed systems over centralised systems for the system developers (but ideally not the users). Needless to say, such an increase in complexity is not without benefits.

2.1.2 Advantages of Distributed Systems

Distributed systems offer users several advantages over centralised systems. Typical examples of these advantages include:

Heterogeneity Support: distributed systems can help overcome problems of heterogeneity in computers, operating systems, programming languages and the underlying networks. Thus users of the system should have a certain amount of freedom in their choice of hardware and software. That is, they should not be constrained to one language, one operating system and one particular machine type. Further, distributed systems can encourage the integration of new technologies.

Fault Tolerance: distributed systems can be designed so they are more tolerant of certain system failures. This could also be considered as a measure of the availability and hence reliability and dependability of the system. Replication of components is one mechanism by which this can be achieved.

Performance Issues: it might well be the case that distributing a system results in increased computing resource usage, and hence higher performance. Migration of programs and data to balance loads between components is one way this might be achieved.

Further Optimisations: as well as system computing performance considerations, distributed systems encourage further optimisations to exploit resource potential. For example, cost optimisations can be addressed.

Distributed systems are not without their drawbacks however. It may not always be desirable to distribute a system. For example, aspects related to information integrity and system security may outweigh performance, heterogeneity and reliability constraints. Perhaps the biggest drawback to distributed systems is their increased complexity. This is especially so for the developers of these systems. This complexity should ideally be hidden from the user of the system, *i.e.* it should be made transparent to the system users.

2.1.3 Complexity Solving Through Transparency Provision

Transparencies offer a means to hide complexity. Complexity in distributed systems arises from their inherent properties as described in section 2.1.1. Several works [4, 107] have identified a core set of transparencies that enable various aspects of distribution to be hidden. For example, ODP [107] supports the following transparencies:

Access Transparency: which masks differences in data representations and invocation mechansims between objects. Thus the problems of heterogeneity in hardware and software should be hidden from users of the system.

Failure Transparency: which hides certain system failures and subsequent recovery from users.

Location Transparency: which hides the location of components so that users need not know where a particular component is.

Migration Transparency: which allows components to be moved around the system without disrupting existing services.

Relocation Transparency: which allows interfaces that are bound to other interfaces to be moved around the system without those interfaces being aware of the relocation.

Replication Transparency: which hides the effects of having multiple copies of a component, *e.g.* for reliability or performance considerations.

Persistence Transparency: which hides from users the possible storing and subsequent retrieval of components from permanent media.

Transaction Transparency: which masks coordination of the behaviour of a collection of components to achieve transaction processing capabilities.

These transparencies are typical of those that might be found in a distributed system solution. They do not represent an exhaustive list however. Other transparencies might relate to performance issues or focus on concurrency issues for example. Similarly, different distributed systems will most likely have different complexity aspects that have to be overcome and hence require different transparency mechanisms (or a subset of these transparencies) to be in place.

Technologies and paradigms for enabling transparencies and for overcoming the increased complexity of distributed systems generally are thus highly desirable. We discuss two in particular: the client/server paradigm and object technology. From these we discuss the role of architecture in the development of distributed systems, and compare and contrast current architectural approaches.

2.1.4 Client/Server Paradigm

Historically, the development of distributed systems relied on the use of conventional programming languages with extensions to handle inter-process communication. To exploit system resources, the programmer had to use low-level operating system routines. Typically, these system resources were organised as clients and servers. Servers were used to provide (unsurprisingly) services which clients could make requests for.

Whilst appealing in its designation of responsibility, the client/server paradigm was only a partial solution to the problems of distribution. That is, whilst it was a technology that allowed distributed communications to be addressed, it did not overcome the problems of complexity inherent to distribution. That is, the problems of distribution and their proposed solution were very much apparent.

This situation was alleviated considerably with the introduction of the remote procedure calling paradigm. This was initially considered by White [208] as early as 1976, but gained wider acceptance in the early 1980's with Nelson's thesis [150]. RPC is a technique to represent the interfaces between communicating components in a distributed system with facilities supported at the language level. Specifically, it offers a programming language independent *interface definition language* (IDL) that can be mapped to specific programming languages running on different operating systems and hardware. Thus, communication between components and their underlying technologies no longer had to be explicitly addressed by the programmer.

Nowadays, RPC is most often used in conjunction with the client/server paradigm. There have been several RPC implementations, both commercially [6, 48] and in research systems [11]. Despite its popularity, it has several shortcomings as discussed in [190]. For example, there are performance issues, the inability to model streams of information and the lack of multicast capabilities. These features are all being investigated, *e.g.* [204, 170] now have features to enable multicast communications, and [4] are looking at how performance aspects and streams of information can be included in the RPC protocol.

Typically an IDL consists of a grammar given in a decidable form such as Extended Backus-Naur Form. With this grammar, structures can be built to represent the syntactic aspects of system interfaces and hence the syntactic aspects of the communication protocol they support, *i.e.* the operations present in the interface to the system that may be invoked by other systems. These operations usually consist of a name, some input parameters and possible output parameters (exceptions). In some of the more recent IDLs [87, 170] a notion of the semantics, *e.g.* best effort, at most once, ... of the operation may also be provided.

IDLs provide a basic set of types that may be used as a starting point for constructing the data associated with communications. For example, Booleans, integers, and strings are often provided and the syntax for constructing more complex types such as records, structures, unions and interfaces are given. From this largely syntactic language, mappings to different implementation languages is then made. Typically, an IDL fragment is specified to represent the syntactic aspects of the interface between the client and server, *e.g.* the operations the server offers and the client can invoke. With this, implementation fragments are produced, *i.e.* basic functions structures that have to be developed (implemented) further. This is often done using a compiler that translates the IDL to a specific implementation language. These basic function structures are commonly referred to as stubs on the client side and skeletons on the server side. Thus, provided the local and remote system support the same communications protocol, *i.e.* they use the same IDL (or can translate between different IDLs) on their respective systems, issues related to heterogeneous hardware and software can be overcome.

The use of an IDL may thus be regarded as a common type language vocabulary with which type checking can be done. Thus type checking the syntactic aspects of two interfaces written in IDL ensures compatibility of the operations they support, *i.e.* the operations that allow access to their systems and the operations that the systems wish to invoke from their environment. Typically compatible operations have the same name and same number of parameters, and these parameters are compatible also. We shall investigate operation names and compatibility between parameters in more detail in chapter 5, 6 and 7.

Procedure calling is essentially a serial language paradigm [15]. Distributed systems are inherently parallel. Thus whilst the client/server RPC approach does have advantages in overcoming problems of distribution, a more flexible approach is needed. A different but complementary approach is that of object technology [27, 46, 165].

2.2 Object Technology

Object technology has emerged in recent years as a major technique in software engineering. This is reflected by the current interest in object-oriented languages such as Smalltalk [83], C++ [189] and Eiffel [137]. Similarly object-oriented design and analysis techniques, such as Booch [27], Coad & Yourdon [46] and Rumbaugh [165], are generating much interest in the computing community. Object-orientation, it could be argued, has pervaded all aspects of computing science: analysis [45], design [46], programming [189], operating systems [10], databases [133] and, of relevance to this thesis, formal methods. We discuss how object-orientation has influenced formal techniques in the following chapter.

Despite the obvious success of object technology today, there exists a wealth of differing definitions of the concepts that underlie object technology. Wegner [205] provides a perspective on object technology that has widespread acceptance. He describes object-oriented systems as those that support the concepts of *objects*, *class* and *inheritance*. An *object* is defined as an encapsulation of a set of methods with state that remembers the effect of those methods. A *class*, he argues, is a template from which an object is instantiated. That is, it defines the behaviour of the object in terms of a list of methods and their implementations. Finally, re-use of code is achieved through inheritance. This is a relationship that exists between classes. Thus one class may inherit from another and subsequently alter or extend the behaviour to meet new requirements. The inheriting class is commonly referred to as the *subclass*. One consequence of inheritance is that an object's methods may belong to its class or to one of its superclasses if it inherits. Hence it is necessary to search the class hierarchy when a method is invoked. This search is normally carried out at run-time and is referred to as *dynamic binding*. It is worth noting here the distinction between binding and typing, and further, the distinction between dynamic and static binding, and dynamic and static typing. Object orientation for various reasons, *e.g.* its widespread interpretation, has blurred these areas somewhat so clarification is necessary. Also as we shall see in chapters 4, 5 and 6 the notions of type and class are fundamental to architectural development.

2.2.1 Type Checking and Binding

Type checking deals with whether or not certain operations are valid. From the perspective of distributed systems we are interested in two main areas of type checking. Firstly, to support interworking we need to perform checking on whether two (or more) interfaces that provide access to given enterprises can, in an abstract sense, be brought together and have some sort of guarantee that they will interwork correctly. Secondly, to support system evolution we need to check whether one interface is an acceptable substitute for another interface. Type checking has traditionally been used to determine whether these are syntactically possible. That is, we require that the operations requested by one interface must exist in the other interface. Similarly, the parameters associated with these operations must be compatible.

There are two main choices as to when these checks should take place: statically or dynamically. With static type checking, type checks are performed at compile time (or link time). The advantage of this are that all type errors can be caught before the program runs. The disadvantage is that all types have to be fixed at compile time which, with regard to open distributed systems where new resources may be found and expected to interwork with existing applications, may not always be possible. An alternative is to adopt dynamic type checking. Here type checks are performed at run time. This has the advantage of enabling a more dynamic programming style since types no longer have to be resolved at compile time. Thus new resources may be found and checks done on whether they can interwork with existing resources without having to recompile running applications. The disadvantage of dynamic type checking is that extra complexity is introduced since possible run time type errors may arise. For example, the classic "message not understood" response has to be dealt with. In certain systems, *e.g.* safety critical ones, this may be an unacceptable option.

It is worth noting here that the "message not understood" paradigm arises from a lack of type checking. That is, type systems are not strictly necessary for object-oriented languages. As stated by Black et al. [13], in value-oriented programming languages the type system protects the user from misinterpreting values, *e.g.* it prevents character operations being performed on integers. In object-oriented languages, however, more protection is offered. That is, values can only be changed inside an object through invoking the operations that the object exposes to the environment. If an attempted operation invocation occurs that

does not have an understood format, *e.g.* an integer was given where a character was expected, say, then the invocation will be rejected.

Binding on the other hand is concerned with resolving which code will be used when an operation occurs. Inheritance offers itself to dynamic binding where the occurrence of a method associated with an object instantiated from one class, might require the class hierarchy to be searched at run time to find the code that implements that operation. The advantages of dynamic binding are that it supports evolutionary programming. For example, classes in the class hierarchy may be changed without necessarily having to recompile programs that use them. The main drawback with dynamic binding is that it incurs a certain amount of overhead in that the class hierarchy has to be searched at run time. Static binding on the other hand resolves all such code look-up calls at compile time (or link time). This means that an operation invocation corresponds to a simple procedure call, and as such there are no run time overheads. The disadvantage of this are that bindings cannot change without recompilation. As a result, it goes against the grain of object-orientation in supporting dynamic, evolutionary systems. Hence with regard to distributed systems, static bindings do not offer a dynamic (flexible) enough mechanism to achieve systems development.

The ability of an object to be substituted for another object is termed polymorphism. This substitutability is one of the major advantages put forward in object-oriented literature [27, 165]. Polymorphism plays a pivotal role in developing type systems that support evolutionary (distributed) systems. We thus provide some background to polymorphism and its importance in object-oriented systems.

2.2.2 Polymorphism and Object-Orientation

Polymorphism is defined to be the ability for a value to have more than one type. Thus polymorphic values can be used in different contexts. A general classification of polymorphism was presented by Cardelli and Wegner [40]. They identified two main forms of polymorphism: *ad-hoc* and *universal*.

Ad-hoc polymorphism works only on a limited number of types in an unprincipled way. Cardelli and Wegner decomposed ad-hoc polymorphism into two areas: overloading and coercion. One example of overloading might be when a single operation is used to execute different code depending upon the parameters it is supplied with. Coercion is used when a parameter passed to an operation has to be converted to one that is acceptable to that operation, *e.g.* an integer was sent but a real was required might entail converting the integer to a real.

Universal polymorphism was also decomposed into two areas: parametric polymorphism and inclusion polymorphism. With parametric polymorphism the same operation will work uniformly on a range of inputs. This is similar to overloading conceptually in that a single operation can be used with several different input types. Parametric polymorphism performs the same function (executes the same code) each time it is requested. As such it is more principled in its approach to achieving polymorphism. Operations supporting inclusion polymorphism also work on a range of inputs. With inclusion polymorphism however, the range is determined by a subtype relationship. Thus an operation supporting inclusion polymorphism that requires a type as input should accept a subtype of that type.

Polymorphism brings with it a great deal of flexibility. It encourages abstraction through allowing abstract operations to be defined over many types. This also lends itself to aspects of behaviour sharing and, in the case of parametric polymorphism, to potential code sharing.

2.2.3 Object Technology and Distributed Systems

With regard to distributed systems, object technology offers at first glance an appealing paradigm within which distributed systems may be reasoned about and developed. The idea of systems and their components being represented by objects that send messages to one another to achieve a common task is appealing due to its simplicity. It is also very much complementary to the classic distributed client/server RPC approach as discussed in section 2.1.4. The ability to encapsulate information and information processing activities also helps to overcome many of the complexities of the system as a whole.

If a single system might be modelled as an object that sends and receives messages, then of particular interest is how this system (object) might interwork successfully with other systems (objects). This is achieved through introducing types into the system. We have seen in section 2.1.4 already that IDL offers one way to construct the syntactic aspects of an interface type system. It offers a means whereby the

potential for successful interconnection (or syntactic interworking) can be established. It is this potentiality that we are primarily concerned with. It should be pointed out however that as presented so far, an IDL is used to develop static type systems. The benefits of static typing are not always a desireable feature. Languages such as Smalltalk [83] have a large popularity since they release the programmer from many of the constraints that static type systems impose, *e.g.* knowing all type information at compile time. These constraints, although offering type safety in that errors will not occur in messages not being understood, are restrictive in that they prevent dynamic evolution. This is especially limiting in distributed systems. We shall see in section 2.3 how certain architectures have attempted to incorporate dynamic typing whilst using IDL as a basis for type checking.

Thus as far as type systems and open object-oriented distributed systems are concerned, there are several issues that have to be addressed before deciding upon a particular typing model. For maximum flexibility, *i.e.* the ability to find resources and interwork with them "on the fly", dynamic binding and dynamic type checking are required. As argued previously, such flexibility is often at the expense of type safety. Hence other typing models may be more beneficial in certain circumstances, *e.g.* use static typing in conjunction with dynamic binding. This enables an approach whereby all messages will be understood but the exact interpretation of these invocations will not be resolved until run time. Whilst static binding has associated with it lower overheads in not requiring dynamic searches for method implementations at run time, with regard to distributed systems such advantages may be overweighed by the restrictions imposed by knowing such information at compile time. Thus static binding prohibits the exact flexibility that distributed systems attempt to encourage.

2.2.4 Tension between Object Technology and Distribution

It should be pointed out that whilst offering many immediately identifiable advantages, assuming that object technology is the panacea for distributed systems development would be myopic. Object technology and its application to open distributed systems raise issues that have to be addressed. For example, inheritance as an approach for code re-use may not be as important when dealing with distribution [15]. Issues related to migration of objects across networks, and the subsequent possibility of separating objects from their implementations (classes), have to be considered. It might be that such issues can be avoided, *e.g.* through copying the class hierarchy also when the object is migrated, or replicating the class hierarchy on several network nodes. These and other possibilities are discussed in more detail in [9].

Further, inheritance can also lead to the implementation hierarchy not being clearly defined, *e.g.* subclasses are not always subtypes. Thus when an implementation changes its behaviour, all related implementations (in the inheritance hierarchy) may subsequently have undefined behaviours. Ordinarily this is not such a problem when the implementation hierarchy is under the control of a defined group who can update related implementations simultaneously. But it is precisely this ability to control and change a set of related components simultaneously that differentiates an application, even a complex one, from a true distributed-object system.

Such issues have lead to some, *e.g.* Wegner [205], stating that "inheritance is incompatible with distribution". This has lead others [152] to come up with different interpretations of object orientation that are more applicable to distributed systems. For example, instead of emphasising objects, classes and inheritance as with Wegner, it is argued in [152] that more useful concepts for distribution are encapsulation, abstraction and polymorphism. Encapsulation here corresponds to the grouping of operations and data hiding. Abstraction is the ability to group entities according to common properties and polymorphism is as defined earlier. This, it is argued, is a generalisation of Wegner's definition which corresponds precisely to Wegner's definition when the mechanisms for encapsulation, abstraction and polymorphism are objects, classes and inheritance respectively.

2.3 Architectural Approaches to Distributed Systems Development

Whilst object technology can help to reduce the complexity of distributed systems, it is itself not *the* solution to developing and reasoning about distributed systems. Rather it provides a means for describing (and building) distributed systems, but does not state how such descriptions can be constructed. This is in many

ways only natural since different systems will almost certainly have different requirements resulting in different functionalities. The problem of developing generic solutions to distributed systems in the large can, to a certain extent, be alleviated through an architectural approach. Typically an architecture consists of a core set of:

- **components:** these may for example be used to provide, either completely or in part, services for overcoming specific aspects of distribution, *i.e.* they offer a way to achieve the transparencies described in section 2.1.3.
- **architectural concepts:** these are more likely to be lower level, more elementary entities, that are used to build abstract representations of (elements of) distributed systems.
- **structuring rules:** describe how these architectural concepts may be composed (structured) with one another to build larger structures (systems). This includes a description of how the structured systems can be configured wih the aforementioned components.

Using the term architecture in the house construction sense of the word, the analogy of the definitions given here is that the components correspond to prefabricated structures that serve a specific functionality (or functionalities), *e.g.* doors. The concepts correspond to the more elementary building bricks, or rather the templates for making the building bricks. The structuring rules correspond to both the cement that holds the bricks together and to a certain extent, the builders imagination. That is, there are many ways to build a house from similar building materials. This last issue has led some [3] to suggest that structuring, or rather more generally design, is close to an art form. The identification of patterns that repeat is now viewed as a possible solution to design and hence possible code reuse. [77] has provided a basic set of design patterns that repeatedly appear in object-oriented software development.

With regard to this thesis, the role of architecture is crucial. Writing specifications of distributed systems can be alleviated greatly through architectural considerations. This issue is discussed in more detail in section 4.1. Recent works [195, 196] have also investigated the role of architecture in a communications context.

Different approaches have been put forward for how the components, concepts and their associated structuring rules associated with distributed systems development are to be represented. The choices made with the structuring rules in particular have direct consequences on the notion of openness given in section 1.1.1. One proposal is to provide bundles of software (components) that can be used as an enabling solution for distributed systems development, *i.e.* this software can be used with different software and operating systems from different vendors to overcome various aspects of distribution.

One such example of this approach is the Distributed Computing Environment (DCE) [170, 204] developed by the Open Software Foundation — now called the Open Group. The DCE consists of an integrated collection of components based on the client/server paradigm. These include the DCE Remote Procedure Call, the Cell and Global Directory Services, the security service, DCE Threads, Distributed Time Service and the Distributed File Service. These components may be used (selectively) by developers to build and manage their own particular applications that require aspects of distribution to be dealt with and overcome.

The DCE was developed largely before object technology received such attention. As a result, the DCE takes a predominantly procedural approach to distributed systems development that works at a lower level of abstraction than others.

A different proposal for developing architectures that has received particular attention is the notion of a framework. Simplistically, a framework provides an organised environment for running a collection of objects. This environment contains certain components that facilitate the software development process. It also provides tools that allow new components to be constructed and to interwork in that framework. Typically, this framework corresponds to a proprietary operating system running proprietary software. Examples of framework based approaches include Microsoft's Object Linking and Embedding (OLE) [36] and Components Integration Lab's (CILab[1]) OpenDoc [198]. Whilst enabling software to be built in an environment that allows a degree of flexibility, *e.g.* communication between applications can be provided

[1] This is a consortium formed in 1993 consisting of Apple, IBM, Novell, Oracle, Taligent, SunSoft, WordPerfect and Xerox.

for directly, such approaches have not as yet addressed issues of distribution in their entirety. Recent collaborations, *e.g.* OpenDoc and the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [87] are actively addressing this issue though. We shall discuss CORBA in more detail in the following section. Microsoft's OLE is also addressing distribution issues. Specifically it is adopting an RPC based approach to enable remote communications to be dealt with, based upon that of the DCE.

A third approach to developing architectures for distributed systems is to provide an abstract architecture. That is, rather than attempt to provide a single solution for problems of distribution, an architectural approach for developing a multitude of solutions is provided. We argue that such an approach is beneficial since different systems will undoubtedly have different requirements and expectations, hence they will subsequently require different features for overcoming problems of distribution. This approach also lends itself more closely to our interpretation of openness as described in section 1.1.1. We consider and contrast two such approaches in particular: the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [87] and the current standardisation activity of Open Distributed Processing [107, 108, 109, 110, 111].

It should be noted here that the approaches mentioned previously are also based on an abstract model. Microsoft's OLE is based on their Component Object Model (COM) [36] and CILab's OpenDoc is based on the System Object Model (SOM) [124] from IBM. These models use ideas from object technology to enable integration between proprietary applications, and as such do not fulfil our interpretation of openness as given in section 1.1.1.

As an aside, it is worth noting that the COM and SOM show perfectly the problems facing object technology in terms of terminology and approach. COM for example does not support inheritance as given classically by Wegner [205], but rather uses a form of containment and aggregation to achieve the same effect. That is, objects encapsulate other objects and messages directed to the encapsulated object are accessed by the containing object and subsequently forwarded. This and other issues relating to the purist approach to object technology are currently subject to much public debate.

2.3.1 Introduction to Object Management Group's CORBA

The Object Management Group (OMG) is a consortium operated as a non-profit company. Its objective is to create a standard for interoperability between independently developed applications across networks of computers [87]. The OMG's member organisations include most of the major information technology vendor companies and many end-user companies. OMG works by adopting interface and protocol specifications within the context of a jointly agreed Object Management Architecture (OMA) [90]. Typically, these specifications are drawn up by collaborating member companies.

OMG has recognised the importance of object technology and from its inception has been a keen advocate of it. The central component of the OMA is the Common Object Request Broker Architecture (CORBA). This consists of three main parts: its object model, its IDL and its Object Request Broker (ORB). The object model introduces objects and their associated concepts. In particular, an object is described as an entity that encapsulates state and provides one or more operations acting on that state. These operations are described using IDL. This IDL is based on ideas from object technology and has a syntax based on C++ [189]. It is worth noting here that the object model put forward does not attempt to deal with composition of objects, nor does it suggest any particular behavioural expectations of the objects. Further, the specification does not explicitly state how many interfaces an object should possess, *i.e.* is it restricted to having only one interface or can it have more? We shall see in section 2.3.2 that other object models do state explicitly that objects can have more than one interface. In CORBA, if a client has a reference to an object then it can invoke any operation associated with that object. This has certain disadvantages that are discussed in section 2.3.2.

The ORB is primarily responsible for handling aspects of distribution. That is, the transparencies described in section 2.1.3, or some of them, will be realised to some extent by an ORB. For example, the possible remote location of an object will be hidden from the accessor (client) of that object when communication is made via the ORB. Similarly, issues in the relocation, migration and tracking generally of objects are dealt with by the ORB.

The main components of the OMA are: the ORB, CORBA services and CORBA facilities. Their relation is shown in diagram 2.1.

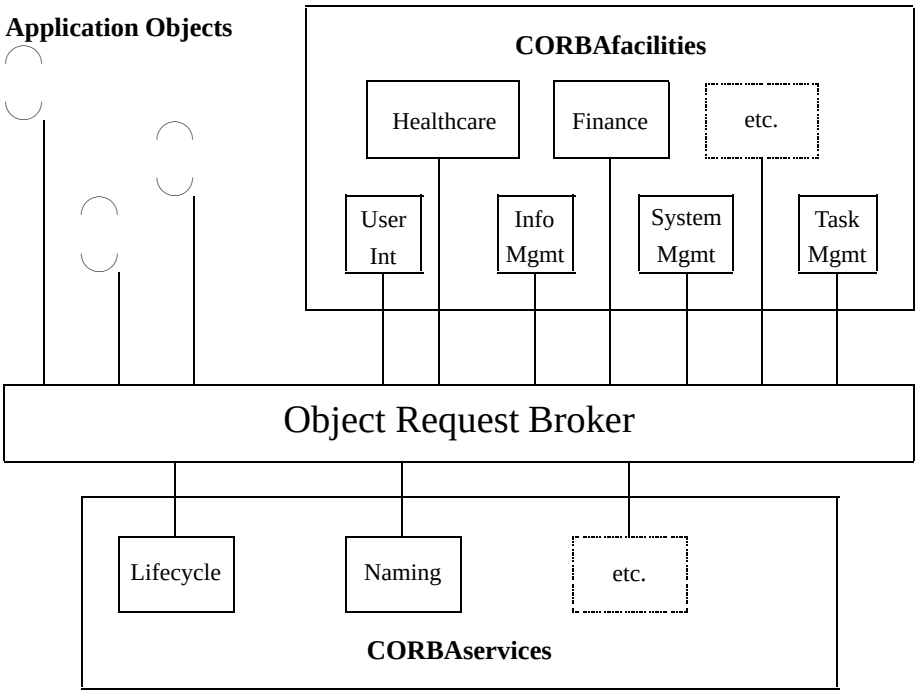


Figure 2.1: Main Components of the OMA

CORBA Services

Object services are basic services that other objects might find useful [89]. They are expected to be available on all ORBs [90]. The OMA has identified the following services:

- **naming service**: is the principal mechanism for objects on an ORB to locate other objects. Names are human recognisable values that identify an object. The naming service maps these human names to object references. Naming hierarchies may then be created and used to manage the object names in the system. We shall see in section 7.2.6 that naming is an essential area that can aid in overcoming many of the problems of distributed interworking.
- **relationship service**: allows entities (CORBA objects) and the relationships that exist between them to be explicitly represented.
- **lifecycle service**: provides operations for creating, copying, moving and deleting objects. This includes operations for handling groups of objects that have been related using the relationship service.
- **persistence or storage service**: provides a set of common interfaces to the mechanisms used for retaining and managing the persistent state of objects. Ultimately the object has the responsibility for managing its own state, however this service may be used by the object should it wish to delegate the work.
- **event service**: allows objects to dynamically register or unregister their interest in specific events. Here an event is an occurrence within an object specified to be of interest to one or more objects. Objects are informed of these events through notifications.

- **transaction service**: provides interfaces to coordinate and manage transaction processing activities.
- **concurrency control service**: provides interfaces to acquire and release locks that let multiple clients coordinate their access to shared resources.
- **trader service**: provides a means for clients to find services that have been exported. In particular, the trading service allows for searches based on some client criteria where detailed information is unknown. We shall discuss trading in some detail in chapter 8 of this thesis when we apply the architectural semantics to develop a trader specification.
- **property service**: provides operations to enable named values to be associated with any object. Using this service, it is possible to dynamically associate properties with an object's state, *e.g.* a title or a date.

CORBA Facilities

CORBA facilities [88] also provide services that other objects might find useful. As stated by Edwards in [51], "*common facilities* can be thought of as an *application toolkit*, whereas common services are an *infrastructure toolkit*". Common facilities are optional on ORBs [90]. Examples of common facilities include:

- **user interface facilities**: include support for the general display and printing of objects, as well as user desktop facilities such as help information.
- **information management facilities**: include support for information storage and retrieval, as well as mechansims enabling objects to interoperate by exchanging data, *e.g.* file formats for data interchange.
- **systems management facilities**: support the management and control of networks and objects.
- **task management facilities**: provide an infrastructure for applications and desktops that model and support the processing of user tasks. This infrastructure contains facilities for managing user and project workflows, rules and communications. High-level messages communicate requests originated by user transactions in a task oriented human interface to task management facilities.
- **vertical market facilities**: are used to support specific markets. These facilities may thus be regarded as being driven by the needs of specific enterprises. Examples of such facilities include accounting and finance, healthcare and image processing.

These user interface, information, systems and task management facilities are typically referred to [155, 171] as horizontal facilities. It is predicted [155] that the CORBA facilities will eventually dwarf CORBA and the CORBA services in size. OMG does not intend to produce all CORBA facility standards itself. Rather, it has put in place procedures so that other consortia standards can be incorporated into the OMA.

ORB Components

In figure 2.1 the ORB is represented by a contiguous block. In reality an ORB is more likely to consist of several logically distinct components. We mention briefly some of these that relate directly to this thesis.

Dynamic Invocation Interface

The dynamic invocation interface (DII) provides clients with an alternative to using stubs generated from IDL when invoking a server. It is used when the detailed knowledge of the operation to be invoked is not known at run time. That is, when stubs for a client are implemented, the servers they have access to are determined statically, *i.e.* when the client is implemented. Thus any new servers added to the system at a later date cannot be accessed directly by the existing client. To overcome this CORBA has introduced the DII. This enables clients at run time to:

- discover new objects and their associated interfaces;
- retrieve their interface definitions;
- construct and dispatch invocations and receive the resulting response or exception information.

To achieve this the client must supply information about the operation to be invoked and the types of the operation arguments. The immediate question is, how does the client find out about such information at run time? This is made possible through the interface repository (IR).

Interface Repository

The interface repository (IR) is a service that provides access to interface definitions a given ORB knows about. It can be used both internally by the ORB and also by applications. In effect the IR creates graphs (Abstract Syntax Trees) from IDL specifications and permits operations to be performed on those graphs, *e.g.* checks whether a particular operation is available in a given interface (using the **within()** operation), or alternatively find out which operation a particular parameter is associated with (using the **contents()** operation). Clients may then browse the IR to find out which interfaces are available. To make this browsing more effective, naming issues are critical, so meaningful hierarchies can be structured and traversed.

One of the problems of the IR as a type information repository, however, is that the type information is entirely syntactic. We argue that the IDL definitions alone do not capture enough information to enable successful interworking between clients and servers to take place. It is of course possible to augment the IDL to include comments which state the expected or desired behaviour of given operations. This approach has, for example, been taken by the Telecommunications Information Networking Architecture Consortium (TINA-C) group in their Object Definition Language (ODL) [156]. Whilst a short term solution to overcoming the problem of relying too much on syntactic aspects alone, this is unsatisfactory for several reasons. For example, it does not lend itself to testing, verification or reasoning in general. Further, natural language as a behaviour description mechanism is inherently flawed. Indeed, this was one of the main reasons for the development of formal languages. We investigate how behaviour is typically represented in formal languages in the following chapter. In particular, we focus on those languages that have been influenced by object technology.

CORBA as an Abstract Architecture

The OMG's CORBA specification has many of the features one would expect to find in an abstract architecture for distributed systems as described above in section 2.3. The ORB, services and facilities are components that overcome aspects of distribution; the object model and IDL are used to describe basic architectural concepts; and the IDL is used to describe how the concepts and components may be structured with respect to each other. With regard to this thesis, this last point is crucial. That is, whilst it is important to have a syntactic means to represent the interfaces between architectural concepts and components so messages sent will be understood, this does not in itself imply that the system as a whole will function correctly. Thus using the definitions in section 1.1.1, this approach lends itself to interconnectivity but not interworking. Using the house construction example again, this means that we may have doors that never open, say — assuming the message passing analogy in our architectural abstraction is equivalent to somebody pushing the door, *i.e.* the push message is accepted but does not have the desired effect. In effect, we have a syntactic structuring mechanism that does not address behavioural concerns.

The consequences of this for CORBA generally are that issues such as conformance to the CORBA specification are not dealt with. That is, whilst it is possible to show that a system is not conformant, *e.g.* a legal CORBA IDL specification is rejected, there are no current conformance tests to check whether a given ORB truly complies to the CORBA specification. Rather, compliance of all ORB products to the CORBA is currently based only on their supporting the OMA object model, IDL and certain components, *e.g.* the IR and DII [155]. In addition the product must support at least one IDL to implementation language mapping. In fact, conformance and testing in general to the CORBA is currently, unsatisfactorily, by vendor assertion.

It should be pointed out that this largely syntactic approach has been deliberately chosen by the CORBA community and is not an oversight. The reason is that extending the structuring to deal with behavioural

and other forms of checking is non-trivial. Further, in developing an abstract architecture for a multitude of distributed systems it is not possible to prescribe the behaviour of specific architectural concepts explicitly. Thus one would expect the door to a bank vault to have different behaviour or characteristics from the door of a greenhouse say, yet they are both manifestations of the door construct. Since our interest lies in specifying distributed systems, we require an abstract architecture that does deal with behavioural aspects. We shall also see that behavioural aspects alone are not enough if interworking systems are to be specified. One abstract architecture that has tried to address these behavioural issues to a certain extent is that of ODP [107, 108, 109, 110, 111].

2.3.2 Introduction to the ODP-RM

ODP represents a major effort between the International Organization for Standardization (ISO) and International Telecommunications Union (ITU-T). This work identifies and attempts to provide a framework for the development of standards for distributed systems. This framework has been set out in a Reference Model of ODP (ODP-RM). It defines an architecture through which distribution, interworking and portability can be achieved.

The ODP-RM recognises that it cannot provide an infrastructure to meet all of the needs of distribution for all applications. Different systems will almost certainly have different demands on the infrastructure. The ODP-RM does, however, provide a framework for describing these infrastructure components and their configuration. Given applications may then select the components they need for their particular concerns. Thus the ODP-RM is in effect a framework for developing standards for distribution, where the standards to be developed reflect infrastructure components needed to overcome problems inherent in distribution.

The ODP-RM is based upon concepts derived from current distributed processing developments and, as far as possible, on the use of formal languages to specify the architecture. The ODP-RM uses an approach based on objects and object-oriented technologies for the reasons given in section 2.2.3. It is divided into four main parts.

ODP-RM Part 1 — Overview and Guide to Use

As its title suggests, this document provides introductory material on the ODP-RM framework family of standards.

ODP-RM Part 2 — Foundations

This document contains the definition of concepts and gives the framework for descriptions of distributed systems. It also introduces the principles of conformance and the way they may be applied to ODP. In effect this document provides the basic vocabulary with which distributed systems may be reasoned about and developed, *i.e.* it is used as the basis for understanding the concepts contained within Part 3 of the ODP-RM. The following categories of concepts have been identified in Part 2:

Basic Interpretation Concepts: these introduce concepts for defining other constructs in ODP. In reality, these concepts are meta-concepts which apply to any form of modelling activity and not just ODP. Examples of these concepts include:

- **entity:** any concrete or abstract thing of interest in the universe of discourse being modelled.
- **proposition:** any fact or state of affairs involving one or more entities of which it is possible to assert or deny that it holds for those entities.
- **abstraction:** the process of suppressing irrelevant detail to establish a simplified model, or the result of that process.
- **atomicity:** an entity is atomic at a given level of abstraction if it cannot be subdivided at that level of abstraction.

Since our interest lies in writing specifications and specification writing corresponds to building abstract[2] models, these concepts are fundamental both to the writing process and also, as we shall see in the following chapter, to the formal languages themselves. We shall see that formal languages offer radically different ways to model systems and have different inherent capabilities for representing these interpretation concepts.

Basic Linguistic Concepts: recognises that all modelling concepts and rules will be expressed in some syntax. This might be graphical or follow some textual format. Thus ODP provides a basic linguistic framework for relating the syntax of a given modelling language to the ODP architecture.

It should be noted that problems may exist in expressing a given architecture in natural language. In expressing the concepts and structuring rules for that architecture, a certain vocabulary must be developed (the concepts) and a grammar for constructing legal phrases using that vocabulary (the structuring rules). Thus in this sense an architecture corresponds to a language. We shall see shortly that indeed ODP has developed specific viewpoint *languages*. The problem is, however, that natural language and an architectural description language often overlap in their vocabulary as the previous footnote shows. To overcome this, typographical conventions may be adopted, *e.g.* using italics for architectural concepts.

Basic Modelling Concepts: contain the concepts for building the ODP architecture itself. That is, these represent the most fundamental modelling concepts that are used as a basis for modelling and constructing the more detailed (prescriptive) concepts and to a lesser extent the structuring rules contained in the ODP architecture. The concepts themselves are based in part on ideas from object technology.

With regard to this thesis, these concepts form the basis for writing specifications. They are considered in more detail in chapter 4 along with how they might (*best*) be formalised in LOTOS and Z. It is worth noting here that the ODP object model advocates that objects can have more than one interface [130]. The primary reasons for this are that different interfaces may support different functionalities, *e.g.* separating management interfaces from general usage interfaces. This in turn allows different access controls to be put in place, *e.g.* restricting access to the management interface. Whilst adding in complexity, the advantages to be gained outweigh the costs [51]. We also note that ODP advocates an object model in which references to interfaces are treated as first class items. That is, interface references can be passed around as parameters and used to access the interface they identify. This offers a very flexible approach for dealing with resource discovery and access. We discuss this issue in more detail in chapters 4 and 5.

Specification Concepts: these concepts relate to the requirements on languages used to describe ODP systems. That is, these concepts represent directly the modelling features that specification languages should possess if they are to be used to write specifications of distributed systems. The concepts themselves are based very much on ideas from object technology.

Since our interest lies in writing specifications of distributed systems, these concepts are fundamental to selecting which formal language we use. These concepts and how they might (*best*) be formalised in LOTOS and Z are considered in more detail in chapter 4.

Structuring Concepts: these are concepts that emerge from considering different issues in distribution and distributed systems. Thus for example issues such as naming and organisational concepts are introduced.

It is stated in Part 2 that these concepts may or may not be directly supported by specification languages. In general this is the case since the concepts describe complex structures and it would be overly restrictive to require that these were supported directly by a given specification language, *i.e.* few, if any, specification languages could support such concepts directly. However, we argue that certain concepts relating to behaviour issues should ideally be supported by specification languages. For example, particular ordering constraints on the actions associated with objects are used to explain how objects can have their own behaviour, independent of their environment. Such aspects are essential when modelling and specifying distributed systems, *i.e.* systems of interacting objects, and as such should be supported by the specification language. We discuss these concepts and how they may be formalised in chapters 4 and 5.

[2] Using the English sense of the word and not the definition given previously.

Conformance Concepts: are concepts necessary to explain the notions of conformance to ODP standards and of conformance testing generally.

In particular conformance deals with showing how a given implementation relates to a standard. This is achieved in ODP through the identification of specific points — *reference points* — at which a given test is to be carried out. The observations that occur at these points are then used to see if the system meets a set of conformance criteria.

ODP-RM Part 3 — Architecture

This document contains the specification of the required characteristics that qualify distributed system as open, *i.e.* constraints to which ODP systems must conform. The main features of Part 3 include transparencies, functions, conformance issues and viewpoint languages.

The transparencies identified by ODP are very much similar to those discussed in section 2.1.3. Functions may be regarded as components that aid in overcoming aspects of distribution. ODP has identified specific functions that can be placed into four groups. These are:

Management Functions: provides facilities for dealing with management of system resources. These may control of the processing activities within a specific collection of interacting objects, *e.g.* deciding whether new activities can be started (spawned).

Coordination Functions: provide support for coordinating system resources. Examples of coordination functions include:

- **event notification function**: for notifying specific objects that given events they were interested in have taken place.
- **group function**: to coordinate the interactions of a particular set of objects.
- **migration function**: to coordinate and control the migration of objects within the system.
- **transaction function**: to coordinate and control transaction processing activities.

Repository Functions: provide support for the storage, organisation and retrieval of information in a distributed system. We note two repository functions in particular since they relate very closely to much of the work in this thesis:

- **type repository function:** which deals with type specifications and their relationships;
- **trading function:** which deals with the exporting and importing of services in a distributed system.

Security Functions: deal with aspects of security. For example, issues related to access control and authentication are considered. Providing functions that deal with such issues is made especially difficult in distributed systems due to their potential remoteness.

As well as identifying these functions, ODP has developed an architecture whereby the functions themselves can be described. This function description mechanism is achieved in ODP through the deployment of viewpoint languages. Through these viewpoints, functions and distributed systems in general can be reasoned about in a uniform and complete manner. In particular ODP has attempted to provide an architecture whereby issues dealing with aspects of enterprise interworking can be addressed.

ODP uses the notion of a viewpoint as it recognises that it is not possible to capture effectively all aspects of design in a single description. Each viewpoint captures certain design facets of concern to a particular group involved in the design process. In doing so it is argued that the complexity involved in considering the system as a whole is reduced. ODP recognises five viewpoints, each with its own associated language:

Enterprise Viewpoint: this focuses on the expression of purpose, policy and boundary for a given ODP system.

Information Viewpoint: this focuses on the information and information processing functions in a given ODP system.

Computational Viewpoint: this focuses on the expression of functional decomposition of a given ODP system, and of the interworking and portability of ODP functions.

Engineering Viewpoint: this focuses on the expression of the infrastructure required to support distributed processing.

Technology Viewpoint: this focuses on the expression of suitable technologies to support distributed processing.

Each viewpoint represents a different abstraction of same original system; however, there is likely to be common ground between the viewpoints. We shall look at these viewpoints in more detail in chapter 5.

ODP-RM Part 4 — Architectural Semantics

This document [110, 111] contains a formalisation of a subset of the ODP concepts. This formalisation is achieved through "interpreting" each concept in terms of the constructs of a given formal language. The formal languages that have been applied so far have been Estelle [100], LOTOS [101], SDL'92 [118] and Z [106]. This formalisation and the ramifications it has on the development of specifications form the basis for the work presented in this thesis.

CORBA vs ODP?

It could be argued that ODP and CORBA are attempting to solve the same problems. That is, both groups are trying to provide solutions to enterprises where distributed processing is likely. Certainly, many of the functions identified by ODP as being necessary to aid in providing certain transparencies are reflected directly by an equivalent CORBA service, *e.g.* transaction processing, or they are provided for through the ORB itself. For example, type repositories in ODP parlance correspond to the interface repositories associated with ORBs in CORBA parlance.

It might be construed from this that ODP and CORBA are working in competition producing independent and potentially incompatible architectures. For developers and users of distributed systems generally, this would be the worst possible scenario since it would lead directly to incompatible systems being developed — one of the main reason for developing distributed systems standards in the first place. However, SC21/WG7 and the OMG have recognised their common goals and are keen to ensure that their architectures are not incompatible. This has resulted in several collaborations and liasons between the two groups. Examples of these collaborations have included the adoption of the CORBA IDL by ODP to specify the syntactic aspects of computational objects. Similarly, the ODP trader standardisation is currently undergoing review by the OMG consortium.

2.4 Summary

This chapter has given a broad overview of the context of this thesis: distributed systems. Their associated advantages and added complexities have been discussed. Approaches for overcoming these complexities have been advocated. This chapter has considered two in particular: object technology and reference architectures. We have contrasted current approaches to describing reference architectures focusing in detail on two: the CORBA and ODP. The limitations of CORBA for our purposes, *e.g.* its lack of consideration for behavioural issues, have been discussed. We shall show in the rest of this thesis how the ODP reference architecture in conjunction with formal techniques offers a means to develop specification architectures suitable for writing specifications of distributed systems.

The immediate question that arises from this is which formal technique should be applied to develop specification architectures for distributed systems? This issue is discussed in more detail in the following chapter.

Chapter 3

Formal Techniques and Distributed Systems

This chapter focuses on formal specification languages (FSLs). We provide arguments as to why formality in software engineering is necessary and why this is especially so when dealing with distributed systems. Given the characteristics of distributed systems presented in the previous chapter, we outline desirable properties of FSLs for dealing with these characteristics and hence for specifying distributed systems. Particular branches of the FSL family are introduced and conclusions drawn on their advantages and disadvantages for describing distributed systems, *i.e.* what features they possess for dealing with the characteristics inherent to distributed systems.

Given the discussion in the previous chapter, our work here is biased towards those FSLs that have been touched by object technology. This is those languages that have been used to specify object-oriented systems, and those developed with the intention of being suitable for specifying distributed systems.

3.1 The Need for Formality

Formality can take many forms. Natural language may be written in a semi-formal style through stylised English (or German, or...). Take any legal document as an example. English is not a formal language as such though for several reasons:

- it can be ambiguous;
- it allows contradictions to exist;
- it can be vague and incomplete, *e.g.* when imprecise terms are used.

>From an architectural viewpoint these issues arise from too many concepts and not strict enough structuring rules associated with them. Most writers and poets would argue that this is a good thing since it allows for a richness in word usage that can be used, for example, to bring different emotions into the writing process. This richness is also the fundamental weakness of natural language for capturing the requirements from which software (and hardware) are to be created. There it is precision and conciseness that are more important as opposed to eloquence and emotiveness.

To overcome this problem FSLs have evolved. These may be regarded as symbolic languages that use unambiguous rules for developing expressions in that language and for interpreting the semantics of these expressions. Typically, the semantics is given in terms of mathematical constructs. For example, algebras, set theory and logics are all used as bases for FSLs.

FSLs are often used as part of a methodology for software development. That is, they serve to capture the initial user requirements precisely and act as an aid in the development process generally. This rather vague interpretation of FSL usage in software development is intentional. Specifications may serve many purposes. These purposes should influence the choice of FSL since, as we shall see in section 3.3, FSLs

have disparate properties that make their deployment in given situations more apposite. For example, some languages are more abstract than others and better for capturing quickly and precisely a given user's requirements. From these specifications, formal reasoning through proving properties of the specification can be used to ensure that the user requirements are, amongst other things, consistent. Often this abstract requirements capture has, through the detailed analysis of user requirements, the potential for uncovering requirement anomalies. Other specification languages, on the other hand, work at a lower level of abstraction and are usually more verbose. Whilst not as directly suited to requirements capture, such languages are usually more suited for implementing and rapid prototyping.

Several advocates of formal methods have tried to explain formal methods by debunking the "myths" that surround them [29, 92]. Typical myths include:

- they can guarantee perfect software;
- their usage increases development costs prohibitively;
- they are incomprehensible to clients.

These and other work [30] do not attempt to show that formal methods are the panacea for software development, or the elusive silver bullet as claimed by some [45]. Rather, they argue that formal methods are a tool that can be very useful for software developers. In this thesis, we concur with this philosophy. We do not state or try to show that formal techniques and languages can be used to develop all aspects of distributed systems. Similarly we do not try to prove all properties that might be associated with them. Instead we use formal languages primarily as a reasoning tool for distributed systems and the issues surrounding their development in the large.

3.2 Requirements on Formal Languages for Distributed Systems

Distributed systems themselves raise many issues regarding the application of formal techniques. The first question that faces any person involved in writing specifications is what is the specification to be used for? Is it to be a basis for requirements capture so that precise user requirements can be obtained, or is it driven more by the lower level system requirements, *e.g.* dealing with how the system is supposed to do things as opposed to what it is supposed to do? The constructiveness of specifications has many ramifications on the role of formal techniques generally. That is, the selection of a given language is more often than not driven by available expertise and tools in that technique as opposed to specific (documented) advantages of that language for that purpose in that domain. The primary reason for this is that whilst formal languages have been applied to a myriad of problems and domains, there has been little in the way of directly comparing the advantages and disadvantages of given languages for particular domains and purposes. There are of course exceptions to this, *e.g.* Fischer et al [73] produced specifications of the ODP trader [115] in LOTOS, SDL and Z with the specific intention of contrasting the approaches. Similarly books which introduce families of specification languages, *e.g.* Turner [194] have compared LOTOS, SDL'88 and Estelle to some extent.

Despite these works, selecting specification languages for new and untried areas, *e.g.* distributed systems, is, by and large, a largely unknown and hence undocumented area. As we stated in section 1.1.3 one yardstick for comparison is to see how far formal languages can be used to interpret domain-specific reference architectures. We investigate this issue in the following chapter, when LOTOS and Z are used to interpret the more elementary aspects of the ODP reference architecture. Another more direct approach is to compare those characteristics of distributed systems directly, and from this to identify desirable features of specification languages. It might be considered that this approach would be more directly beneficial to selecting languages for describing distributed systems as opposed to having to develop specification architectures. We argue that the two approaches are complementary. Specification architecture development allows languages to be considered in a more methodical way. For example, lack of certain crucial concepts and how they might be included in language extensions can be analysed in a structured fashion. Direct feature comparison on the other hand allows for immediate suitability considerations to be made on a broad level. Hence we provide an outline of the particular constraints that the characteristics of distributed systems discussed in section 2.1.1 impose directly on FSLs.

Remoteness: from a formal perspective, remoteness of components is usually abstracted from. This is unsurprising since writing specifications is generally founded upon making abstract models. The real-world location at which components exist is not normally part of the model. The notion of location is critical to distributed systems however [85], since it reflects directly on a multitude of issues. For example, components can only interact and systems subsequently interwork if they are aware of one another, *i.e.* know the location and existence of one another. Typically the location of components is reflected in terms of interaction points, *i.e.* locations at which components and their associated behaviours may be found. Formal languages should possess facilities for modelling interaction points should remoteness issues be of interest. This modelling may be done implicitly, *i.e.* it is a feature inherent to the language, or explicitly, *i.e.* it is a feature that the language can be used to model. We shall see how LOTOS and Z offer contrasting ways of dealing with the modelling of interaction points in sections 4.2.10 and 4.2.10 respectively.

Heterogeneity: the area of heterogeneity in languages, operating systems and hardware has to a large extent been overcome by the adoption of interface definition languages [172]. It would be naive to develop an approach that attempted to ignore or undermine the current technologies and solutions for addressing heterogeneity. From a formal viewpoint we argue that a specification language should support reasoning and specification of the syntactic aspects of systems. For example, supporting reasoning about interface signatures (see section 4.3.12) should be possible. Formal techniques should extend the syntactic considerations to deal with behavioural and non-functional aspects of systems also. We shall see in chapters 5, 6 and 7 to what extent LOTOS and Z can be used to realise this.

Concurrency: as distributed systems are inherently concurrent, *i.e.* individual components can have their own independent behaviour, formal languages should possess features for modelling and reasoning about these behaviours. Further, given that distributed systems are typically represented as collections of interacting and interworking sub-systems, formal languages should possess features for reasoning about these interactions. These features may be inherent to the languages, *i.e.* built into their semantics, or can be modelled within the language itself. As we shall see, certain languages that are not intrinsically concurrent can be applied in such a way that issues related to concurrency can be addressed.

It is worth pointing out here that certain work, *e.g.* [98] has argued against introducing concurrency into specifications in the first place. The argument used is that concurrency does not deal with what the system is supposed to do, but with how it is supposed to do it. Similarly, other potentially implementation influencing specification approaches, *e.g.* sequencing of operations, should not be specified. Whilst the basis for the arguments from a purist view is true, *i.e.* a specification represents what not how, we regard this philosophy as myopic in terms of specifying distributed systems. That is, issues related to the specification of collections of interworking systems should deal with issues such as concurrency. It is through consideration of such issues that reasoning about the system as a whole, *i.e.* the *what* become more apparent. It should be pointed out that the arguments above were based on software development in general and not directed at distributed systems. The structured development methodology of software design and requirements of distributed systems are not always well matched. That is, distributed systems rarely, if ever, start completely from nothing, *i.e.* legacy systems typically exist. Further, due to the complexity of distributed systems, specifying the whole system is likely to be overly complex and offer few advantages. Therefore elements of distributed systems are usually specified, *e.g.* sub-systems of some larger system. Through considering these sub-systems in isolation and providing features for reasoning about their combinations, the complexity issues of distributed systems can be approached methodically and systematically. Since concurrency represents one of the main issues in combining sub-systems, we argue that it should be a feature offered by FSLs for describing distributed systems.

A further argument against the work of [98] is that lower level specifications, *e.g.* those dealing with concurrency issues, lend themselves more readily to the software development process in general. We deal with this issue again shortly in considering whether specification languages should be executable or not.

Partial Failures: the notion of (truly) unexpected failure is impossible to specify. By its very nature, if it was a possible expected behaviour then it would have been specified. It is quite possible to specify expected or predicted failures however. Also, it is these that distributed systems attempt to provide facilities for dealing with. Thus a formal language should, for example, be able to model successful and unsuccessful attempts at interactions between components.

Asynchrony: given that we have advocated concurrency, formal languages should allow for the modelling of interactions between systems. Further, it should be possible to specify different forms of interactions between components. That is, the time difference between the sending to the subsequent arrival of messages may or may not be regarded as instantaneous. Synchronous and asynchronous communications should thus ideally be features of the formal language.

Autonomy: whilst it is not necessarily the case that formal languages should support the notion of autonomy directly, it should be possible to specify the behaviour of independent systems. That is, formal specification languages should support some form of modular structuring mechanism. These modules should allow a form of autonomy, *e.g.* the modules decide for themselves how they respond to messages that are sent to them. Similarly, these modules decide for themselves what information they wish to share with the other modules. Object-orientation and the concepts associated with it, *e.g.* encapsulation and interface, provide a suitable platform from which issues related to autonomy can be addressed.

Evolution: formal techniques should be able to model evolutionary systems. This might include extending the functionality of a given system or replacing one part of the system with some other sub-system. Using type theoretic terminology, issues related to inclusion polymorphism and subtyping/subclassing generally are of interest. These in turn require checks on typing in general, *i.e.* if a system is replaced with another then what checks are necessary to ensure that this is a valid replacement in terms of the previous interactions of the old system. Thus subtyping as a mechanism for system replacement is only valid if previous interaction patterns, *i.e.* other systems that interacted with the older system in a certain way, can equally well interact with the replacement system. Here "equally well" typically means that they cannot distinguish between the two and hence their behaviours and the behaviour of the system as a whole is not adversely affected by the replacement.

Mobility: formal techniques should support mobility. This means that it should be possible to formally model sources of information, objects say, and provide features for reasoning about their possible relocation and the subsequent system reconfiguration. That is, formal languages should ideally allow for dynamic communication paths to be set up between the existing components.

As well as these immediate requirements on formal languages for addressing particular characteristics of distributed systems, several other aspects should also be considered when selecting a formal language.

Abstraction: distributed systems are inherently complex. To aid in overcoming this complexity, formal techniques should allow different levels of abstraction to be achieved. That is, they should possess properties that enable, at a given level of detail or interest, various irrelevant aspects of the system as a whole to be hidden. For example, if an automated teller machine were to be specified then it should not be necessary to specify in great detail all aspects of the bank with which it is associated, *e.g.* its overseas investments. Rather, the user interface, the money in the teller machine and the interface of the teller machine to the bank are of primary concern. Formal languages should allow for the selected omission of certain informations to be achieved.

Determinism: a system is deterministic if it is possible to accurately predict its future behaviour through knowing the behaviour of its environment. Non-determinism arises when a system can have differing behaviours regardless of the behaviour of its environment. Typically non-determinism is used to represent internal choice within a system. In distributed systems, non-determinism can be used to address issues such as autonomy, *e.g.* a given sub-system (or object) decides how it should react to interactions with the other sub-systems (objects). As such, formal languages for modelling distributed systems should provide features for specifying and reasoning about non-deterministic behaviours.

Executable versus Non-Executable: there has been much discussion in the formal methods community on whether formal specification languages should themselves be executable. It has been argued by Hayes and Jones [94] that formal specification languages should be non-executable. Their argument is based on four main points:

- requiring specification languages to be executable restricts their expressive power. Thus the specification should be phrased in terms of required properties of the system as opposed to the algorithmic details by which these properties can be achieved.
- proving properties about a specification is a much more powerful method of validation than generating test cases from a specification.
- executable specification languages can unnecessarily constrain the choice of possible implementations.
- it is easier to verify that an implementation meets an abstract specification than to match an executable specification against an implementation for which possibly different data and program structures have been chosen.

An opposite viewpoint has been taken by Fuchs [76]. He argues that non-executable specification languages can be made executable on almost the same level of abstraction and without essentially changing their structure. Further, he shows how the same level of expressiveness can be achieved in executable as in non-executable declarative specification languages.

There has been no real agreement in the formal methods community on this topic. Specifications as abstract reasoning tools versus specifications as abstract implementations both have their relative merits. Certain languages, *e.g.* Z, were not designed to be executable. Other languages, *e.g.* SDL, were designed very much with tool support in mind. It would be naive to expect to implement directly all requirement statements that might be made and hence possibly formalised. Sometimes statements are too general or are impossible to implement directly. Nevertheless the ability to implement or prototype initial abstract specifications is appealing. We shall see how LOTOS and Z offer sample languages from both sides of this argument. We note that object orientation has attempted to provide a unifying framework in which analysis through to design and implementation can be achieved.

Non-functional Aspects: much recent interest in the formal methods community has arisen as to how formal languages might be used to specify non-functional aspects of behaviour [136, 167]. Thus, the specification of quality of service and other constraints that can influence behaviour generally are of interest. It has been recognised [109, 104, 105] that behavioural specification alone is not enough to build reliable and interworking systems. There are more constraints that have to be considered. The overall correctness of a system must satisfy all constraints that apply to it. For example, if functionality and response time are the constraints of interest, then a system that produced correct answers after certain deadlines have been passed would be as unacceptable as a system that produced incorrect answers within a given deadline. Hence, formal languages should ideally be able to model non-functional aspects related to behaviour.

Modelling non-functional aspects places further constraints on the formal languages. There are numerous sorts of non-functional aspects that might be of interest: temporal constraints, costing constraints and constraints related to location are just three examples. The ability to model such features and be able to reason about them in a formal language is highly desirable. Further, modelling and reasoning about such issues should be done in a manner that does not imply a unique implementation.

Tool Support: selection of a formal language may well be influenced by the tool support available for that language. Examples of such tool support might be syntax editors, checkers, simulators and theorem provers. Availability of such tools can to some extent be regarded as an indication of the stability of a given language.

3.3 The Formal Language Spectrum

Broadly, speaking formal languages can be put into one or, as we shall see later in chapters 5 and 6, one or more of the groups as shown in figure 3.3.

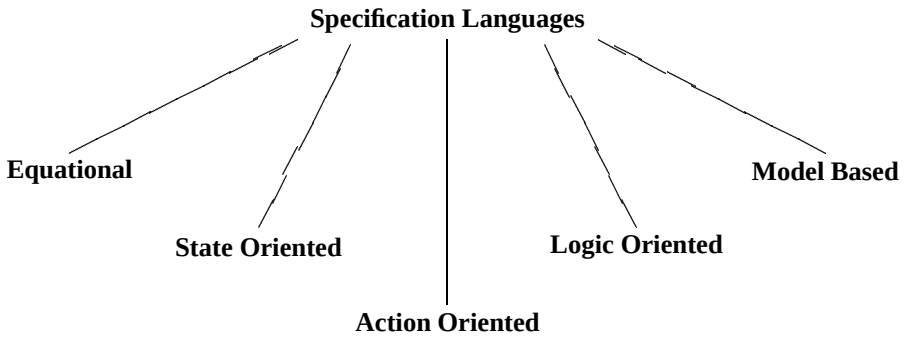


Figure 3.1: The Formal Language Family

We note here that this classification is not exhaustive. Other classifications are possible, *e.g.* synchronous languages. We also note that our intention here is not to provide a detailed account of all aspects of the specification groups given. Instead, we focus on the properties of languages found in these groups and how useful these properties are for modelling and reasoning about distributed systems given their characteristics described in section 2.1.1.

3.3.1 Equational Specification Languages

Equational specification languages concentrate on defining *types*, where a type here corresponds to sets of values called *sorts*. *Operations* are provided on these sorts. These operations give the syntactic features of the sorts, *i.e.* how they can be created, accessed and used in general. The semantic (behavioural) aspects of the operations are defined through writing equations. These equations state equivalence classes between the expressions formed from operations (terms). If the equations are confluent and terminating, *i.e.* Church-Rosser, then the terms can be reduced to a normal form through rewriting. For example, if a push operation on a queue is followed by a pop operation on the same queue, then the equations should be able to deduce that the final queue is the same as the initial queue.

One equational specification language particularly relevant to this thesis is Act One [65]. This is the data typing language used in LOTOS. Act One was developed in 1983 by the ACT-group[1] at the Technische Universität Berlin. The language itself contains four concepts for defining and structuring a system:

- **enrichment:** allows types to import other types in their own definitions.
- **renaming:** allows types to be copied and have different sort and operation names.
- **parameterisation:** allows types to be defined generically. Thus for example, a queue data type might be generic in that it can be instantiated (actualised) to hold different types, *e.g.* queues of integers or characters.
- **actualisation:** instantiates generic (parameterised) types with actual values.

[1] Algebraic specification techniques for the correct design of trustworthy software.

With regard to supporting object-orientation, Act One has certain features that make it useful for specifying object-oriented constructs. For example, sorts have many of the features that one would expect to find in objects. That is, they have an interface given by the operations on that sort whose behaviour is determined by the equations associated with those equations. Further, they support certain forms of polymorphism [40] through parameterisation and actualisation.

Act One does have limitations on specifying object-oriented systems. In Act One, instances of sorts are simply values that have no real state as such. Other algebraic languages such as OBJ [82] have addressed this through introducing the concept of subsorts. In addition, sorts do not lend themselves to inclusion polymorphism [40]. That is, type checking on sorts is done by name equivalence and not any form of structural similarity. As a result, sorts do not support subtyping as such. We shall see in chapter 7 how Act One can be specified in such a way that these subtyping issues can be overcome.

With regard to distributed systems generally and the requirements they impose on FSLs, Act One does not by itself satisfy many of the requirements. This is primarily due to Act One normally being used as a data typing language for use with other languages, *e.g.* LOTOS and SDL'92.

3.3.2 State Oriented Specification Languages

State oriented specification languages describe systems in terms of graphs, where graph nodes correspond to system states and graph arcs correspond to system transitions. With state oriented specification languages, particular emphasis is placed on graph nodes. Examples of state oriented specification languages include: finite state machines; Estelle, SDL'92 and Petri Nets. Estelle and SDL'92 have also been applied to develop an architectural semantics for ODP, hence we introduce them briefly.

Estelle

Estelle (Extended Finite State Machine Language) [100] is a formal description technique (FDT)[2] developed by ISO and accepted by ITU-T, primarily to aid in the formal specification of communication protocols and services. Estelle is based on extended finite state automata. A system is modelled through a hierarchically structured set of module instances, communicating in an asynchronous fashion via the exchange of messages on channels. The syntax of the language and the definition of data types and variables are based on ISO Pascal.

In an Estelle specification, the externally visible interface of a module is defined in the module header, whilst the module body describes the internal structure and behaviour of the module. Module instances define interaction points through which they can send and receive messages. Two interaction points can be connected if they have been defined with opposing roles on the same channel definition. A channel definition contains two roles for the respective ends of the channel. For each role it defines the messages that can be sent. An interaction definition consists of a name together with a set of parameters. To each interaction point, a queue is assigned in which incoming messages are stored. A module instance can also have a common queue that is shared by several or all interaction points. Concurrency is achieved in Estelle through the interleaving of transitions.

The structure of an Estelle specification is dynamic, *i.e.* module instances can be instantiated and released and interaction points can be connected and disconnected dynamically. Each module instance can instantiate or release child module instances, or connect or disconnect their interaction points. The only way for a module instance to access sibling instances (or other module instances which are not its own children) is via exchange of interactions on channels.

SDL

SDL (Specification and Design Language) [118] is an FDT standardised by ITU-T and accepted by ISO. The main use of SDL has been in the telecommunications industry. There are two main representations of SDL: a graphical representation (SDL/GR) and a textual representation (SDL/PR).

[2] It should be noted that the acronym FDT is frequently applied to formal methods generally. This is incorrect. The FDTs are designated standards developed by ISO and ITU-T, namely LOTOS, Estelle and SDL'92.

Like Estelle, SDL is based on non-deterministic extended finite state machines. The data typing part of SDL is based on ACT ONE [65] and ASN.1 [99]. An SDL specification consists of a collection of *blocks*. Blocks can be decomposed into sub-blocks and eventually, *processes.* Processes are represented by non-deterministic EFSMs. Each process has exactly one input queue, the input port. Processes themselves exhibit sequential behaviour. The system behaviour as a whole is represented by an ensemble of independent parallel processes that communicate asynchronously using *signals*. These processes may dynamically instantiate other processes. Signals are sent via *channels* connecting blocks and via *signal routes* connecting processes. There is no priority among the signals sent. That is, they are read from the input ports in the order in which they were sent. Signals from independent processes may arrive in any order. Signals arriving at the same time are enqueued non-deterministically. SDL also does not allow for data to be shared. However, sometimes shared data may be simulated.

In 1992 CCITT issued a new release of SDL that contained numerous new features. These included concepts to enable object-oriented programming and new structures to provide non-determinism.

3.3.3 Action Oriented Specification Languages

As with state oriented specification languages, action oriented specification languages describe systems in terms of graphs, where graph nodes correspond to system states and graph arcs correspond to system actions (transitions). With action oriented specification languages, however, particular emphasis is placed on graph arcs.

One particular example of action oriented specification languages is the *process algebra*. These are particularly well suited to modelling and reasoning about concurrent, communicating systems. We focus on the three most common: Communicating Sequential Processes (CSP) [96]; the Calculus of Communicating Systems (CCS) [141] and LOTOS [101].

CSP and CCS

Hoare's CSP first came to prominence in the 1970's. Originally it was developed as a programming language inspired by Dijkstra's guarded command language and Pascal. Following its first introduction it was developed further, becoming more oriented towards specification rather than implementation, culminating in its definitive process algebraic form published in 1985 [96].

Milner's CCS was originally published in 1980 but underwent a series of modifications until the definitive form appeared in 1989 [141]. Both CCS and CSP were developed primarily with the intention of being suitable for specifying and reasoning about concurrency. Both have the notion of actions[3] and processes[4]. Actions are composed using various operators, *e.g.* prefixing and choice to form more complex behaviours. Actions are atomic, *i.e.* they occur at a single indivisible point in time. In the basic process algebras like CCS and CSP this point in time is undefined. That is, the temporal ordering of actions is given but not the *real* time at which actions occur. Actions may be observable or internal. An observable action requires synchronisation with its environment to occur. An internal action requires no synchronisation and is frequently used to represent non-determinism and spontaneity.

Processes represent structural units for grouping actions. The grouping itself is used to define particular ordering constraints on the actions. Processes are typically composed with one another to form the behaviour of the system as a whole.

Equivalence relations between processes based on the actions associated with them and their observability have been developed for CSP and CCS. We discuss these briefly in the following section on LOTOS. Both CSP and CCS were largely academic developments. LOTOS [101], on the other hand, was developed in response to a practical need. It is based largely on CCS and CSP in conjunction with the data typing language Act One.

[3] Referred to as events in CSP.
[4] Referred to as agents in CCS.

LOTOS

LOTOS [101] was developed by the International Standardisation Organisation (ISO) in the 1980's, becoming a full international standard in 1989. It is partly based on CCS and CSP, although there are significant differences both in terminology and technical details. For example, CCS only supports bi-party interactions whilst LOTOS and CSP support multi-party synchronisations. An overview of these differences and an introduction to the languages generally is given in [71]. We provide an overview of the LOTOS language in appendix C. Further introductory material to LOTOS may be found in [23, 131, 194].

As with CCS and CSP, LOTOS has the notion of atomic actions (events). These are composed to form *behaviour expressions*. LOTOS has a set[5] of composition operators for composing actions. These include: action prefixing; choice; parallel composition; interleaving; disabling and enabling.

LOTOS has a basic form in which actions are solely represented by gates, and a full form in which actions and processes may have data associated with them. This data is defined using the Act One language. When full LOTOS is considered, event offers may contain a set of arguments. These arguments are values of Act One sorts preceded by either a *!* or a *?*. In the former case a single value is given. In the latter case a range of values is given. This allows for different synchronisation possibilities. Synchronisations may only occur when the associated event offers are compatible. This compatibility means that either the event offers themselves are identical, *e.g.* they have identical values for the *!* arguments, or that a non-empty set of values for the *?* arguments exist. Thus event offers with *!* arguments and those with *?* arguments may synchronise provided the *!* value is in the set of *?* values. This form of synchronisation is known as *value passing*. If two (or more) event offers have an *?* argument of the same sort then synchronisation is known as *value establishment, value negotiation* or *value generation*. For example, *g ?x: Nat* $\|$ *g !3;* synchronise and the value of *x* is set to 3. Similarly, if *g ?x: Nat* $\|$ *g ?y: Nat;* synchronise then the value of *x* and *y* is a natural number that is non-deterministically established. Event offers using *?* arguments may restrict the range of values the argument may take through selection predicates. For example, the synchronisation of *g ?x: Nat[x>3];* $\|$ *g ?y:Nat[y<5];* sets the value of *x* and *y* to 4.

As well as immediate synchronisation possibilities, the fusion of the process algebra and Act One also allows events to be guarded so that their occurrence is not solely based on the willingness of the environment to synchronise on events. Information (state) internal to a process can be used to determine whether an action can occur or not. Typically this is achieved with a Boolean expression using Act One values.

LOTOS has a rich set of equivalence relations [35, 125]. These relations identify sets of behaviours that are in some sense equivalent to one another. These identifications typically relate behavioural properties of the processes. Examples of such equivalence relations include: trace equivalence, bisimulation relations and testing equivalence relations.

As well as these equivalence relations, other useful relations have been developed for LOTOS. These relate an abstract specification to a more implementation oriented specification. Examples of these relations include: conformance, reduction and extension. We shall investigate these relations in more detail in the following chapter.

As shown LOTOS is particularly suited to specifying complex behaviours where the temporal ordering of actions in the behaviour is of primary importance. As we shall see in chapter 6, it is often the case that a temporal ordering of actions alone provides insufficient modelling facilities, *e.g.* when real-time or other constraints on the behaviour are given. As a result, there has been a large number of proposals to extend action oriented specification languages. We provide a brief overview of these now.

Action Oriented Specification Language Extensions

There has been a wealth of research into extending process algebras. A large body of work has been generated in timed extensions to process algebras. With regard to timed-extensions to LOTOS some of the works include: [126, 112, 202, 162, 138]. An overview of timed process algebras is given in [153].

We discuss briefly some of the issues involved in introducing time into process algebras and how they have been addressed by extensions to LOTOS.

[5] Unlike LOTOS, both CSP and CCS allow new operators to be defined.

- **modelling time progress**: it is possible to model the progress of time local to a process, *e.g.* so that processes can have different speeds. It is more commonly the case that global time is adopted though. This is more natural given that time in reality has a global meaning.
- **modelling time**: time may be modelled either discretely or densely. In a dense time domain it is always possible to find a value in between a non-equal pair of time values. Typically, real numbers are used to represent dense time. In a discrete time domain, time is divided up into units. Time passage is in multiples of these units. Natural numbers are often used to represent discrete time units. Most LOTOS extensions have adopted a discrete time model, *e.g.* CELOTOS [202], TIC [162], LOTOS-T [138]. ET-LOTOS [127] places no restriction on the time domain though, *i.e.* dense time and discrete time models are supported.
- **timing of actions**: the elapse of time between actions is usually modelled in two ways. The time consumed between action occurrences or the time consumed by the actions themselves. In the first case, actions are given a timestamp when they are to occur. In the second case actions are assigned a time value for their duration. Of course, actions with durations are non-instantaneous.
- **maximal progress**: the maximal progress assumption states that internal actions occur as soon as possible before any passage of time. With this assumption, the occurrence of observable actions cannot be enforced, *e.g.* if the time for their occurrence has been passed, and the occurrence of internal actions cannot be delayed unnecessarily. LOTOS-T [138] and ET-LOTOS [127] are two LOTOS extensions that incorporate maximal progress.
- **action urgency**: an alternative to maximal progress is action urgency. Action urgency can be applied to any event — not just internal events. With action urgency processes may block the passage of time, *e.g.* the action occurs before delays are permitted. It should be noted that if urgency is applied to an event and the environment does not wish to synchronise on that event then deadlocks will occur. U-LOTOS [25] and TLOTOS [126] are two languages that incorporate action urgency.

There have been numerous other approaches to extend process algebras. Probability has been introduced, *e.g.* Pb-LOTOS [136] and LOTOS-TP [139]. McClenaghan's work [136] also included extensions to deal with prioritised actions as does Schieferdecker [167] in her work on structured actions.

The current standardisation activity of Extended-LOTOS (ELOTOS) [112] has considered many of the above topics. In addition, this work is addressing issues such as:

- the modularisation of LOTOS;
- the provision of a new data typing language;
- the typing of gates;
- the provision for dynamic configurations of processes.

As we shall see in chapters 4 and 5 these last two points are particularly relevant to this thesis. The typing of gates allows for gates to specify what parameters can be associated with them. In LOTOS, all gates are polymorphic. That is, they can have any parameters attached to them. Through typing of gates, the parameters attached to gates can be restricted. As a result, it is possible to reason about the syntactic[6] aspects of processes in separation from the behavioural aspects. Further, gate typing can also be used for the detection of certain classes of deadlocks also. For example, the deadlock that result from **g !true;** || **g !0;** could be detected statically if gate typing were used. Discussions on the introduction of gate typing into LOTOS are presented in [78].

The E-LOTOS work is also addressing dynamic configurations of systems. As discussed in section 3.3.3, gates in LOTOS are used for communication between processes. These gates are assigned statically to processes. It is not the case that these gates can be passed around between processes. As a result, LOTOS specifications are not well suited to modelling systems, *e.g.* distributed systems, where new patterns of communication are made possible. Nevertheless, we shall see in chapter 8 how LOTOS can be used in such a way that this dynamicity is made possible through the use of Act One sorts in event offers.

[6] or as we shall see in section 4.3.12 the signatures.

3.3.4 Logic Based Specification Languages

In general, logic is concerned with the study of valid reasoning or arguments. It defines structures that enable us to represent knowledge that would normally be expressed verbally. Logic based specification languages are arguably the most abstract of the specification languages. They exist in many different forms, the simplest being propositional logic. Propositional logic is closely related to Boolean algebra in that propositions have the value of true or false.

Propositional logic only allows variables to have values true or false and does not allow quantification. Predicate logic extends propositional logic with variables. First order predicate logic allows for quantification over variables.

As a specification approach, logic based languages offer a means to write concise and precise statements about the system under consideration. Typically, logic specifications are terse. Of particular interest are modal logics. These may be regarded as logics extended with features for expressing properties that a system should exhibit. We consider briefly two such logic families: temporal logics and process logics.

Temporal Logics

Temporal logics are specialised logics for expressing time-related properties of systems. Temporal logics can be dated back as far as the ancient Greeks, but considerably more recently, interest has grown through the work of Manna [135] and Pnueli [160]. Basically a temporal logic is a logic extended with operators for reasoning about the times at which events occur. Typically, such logics are used to reason about three classes of properties:

- *safety properties:* properties which are true at all times in a system's behaviour.
- *liveness properties:* properties which must eventually become true at least once as the systems behaviour develops. Safety properties are satisfied by systems that do nothing. Liveness ensures that systems do something useful.
- *fairness conditions:* if a certain action is possible then it must eventually occur.

Different temporal logic operators have been defined such as *always* or *henceforth*, *next*, *eventually*. Often these are denoted symbolically, *e.g.* $\Box$ and $\Diamond$.

Process Logics

Process logics allow statements about the properties that process algebraic specifications should satisfy. Milner in his book on Communication and Concurrency [142] presents a simple process logic based on the notion of possible actions. In this logic he presents operators to express the *possibility* and *necessity* of certain events. For example, the possibility of the system performing some event and reaching a particular state in which a certain property is true.

3.3.5 Model Based Specification Languages

In model based specifications an abstract theoretical model of the system to be built is specified. Typically these models focus only on what the system is supposed to do as opposed to how it is supposed to do it. Through these abstract models, reasoning is possible about the system to be developed. Typically this allows for proofs about the system in terms of the abstract model built.

With regard to distributed systems, model based languages have both advantages and disadvantages. Amongst their advantages are that they encourage abstractions to be built. These abstractions may suppress many of the complexities involved in achieving levels of distribution. This abstraction is also especially conducive to capturing high-level requirements of given systems as opposed to other more constructive specification languages, *i.e.* languages that through their usage imply, or possibly constrain, future implementations. For example, they may adopt particular structures or algorithms.

This issue has lead to some to regard model based specification languages with some scepticism. That is, since they generally work at such a high level of abstraction, their role in software development as a

whole is frequently questioned. Thus their abstraction does not lend itself to rapid prototyping of software in general.

Despite this, model based specification languages are one of the most popular [158] specification techniques used today. We focus now on the most popular of these languages: Z. Given our interest in Z and object-orientation, we focus in some detail on approaches to specifying object-oriented concepts in Z and give a brief overview of the more popular object-oriented extensions to Z.

Z

Z is one of the most popular specification techniques used today [158]. It is based on first order predicate logic and set theory. It is argued in [158] that this simplistic mathematical basis for Z is one of the main reasons for its popularity.

A Z specification usually consists of a collection of Z fragments with accompanying explanatory text. The Z fragments are usually given in a definition before usage ordering. This is especially true if tool support [157] is used.

The primary structuring mechanism used in Z is that of the *schema*. This provides a means to group variables whose types have been defined elsewhere in the specification (or are given in the mathematical toolkit associated with the Z language, *e.g.* natural numbers) and to associate predicates with them. The predicates are often written below a dividing line, however horizontal schema definitions are possible also; they are normally used when the schema text is small.

The declaration of a schema introduces a type: an unordered, named collection of various other previously declared types. In some sense a schema type is similar to a Cartesian product, except that the Cartesian product has a specific ordering. Schema names (schema types) are global in scope throughout the specification. The components inside a schema are local in scope to that schema however. We discuss the consequences of the adoption of a schema type as a signature-only typing mechanism in sections 4.2.2 and 7.5.

As well as a specification structuring mechanism, schemas are used to model things that occur in the real world, *e.g.* actions. This is achieved by including in the schema a copy of the variables whose values are expected to change. This copy is primed by convention. Predicates are then used to relate the unprimed variables (the variable values before the occurrence of the schema) to the primed variables (the variable values after the occurrence of the schema). Such schemas are typically referred to as *operation schemas*. Schemas that do not relate primed and unprimed variables are typically referred to as *state schemas*.

As well as the schema, Z also provides axiomatic descriptions to introduce variables, possibly with associated predicates. These variables are global throughout the specification, thus the names they introduce must be new to the specification. If no predicates are given in the axiomatic description then the specification is termed *loose*, *i.e.* there are no constraints that the variable might have. It is also possible to separate the declaration of a variable from the predicates that apply to it, *i.e.* a predicate can appear on its own. This practice is deprecated, however, since it more often than not simply confuses the specification reader.

We provide a brief overview of the Z syntax and mathematical toolkit in appendix B.

Extensions to Z

Numerous works have extended Z in various ways. Of particular relevance to this thesis are those works on object-oriented extensions to Z. We provide an overview of these extensions here. A more detailed comparison of the different extensions can be found in [186] and [123].

As identified in [172], objects provide a useful modelling paradigm to manage complex behaviours as might be found in a distributed system. This relies on the ability of objects to encapsulate information. Z is not object-oriented, however using the definition presented by Wegner [205]. Rather it might be considered as object-based using Wegner's classification. That is, Z can be used to specify a state (via a schema for example) and a behaviour (as operation schemas). However, Z does not support the grouping of operations on a particular state, other than possibly through a textual statement. Thus it is possible to specify objects, but classes are not supported and inheritance of operations is not supported.

Most extensions to Z have attempted to remedy this situation by extending the Z notation to include a new structuring mechanism: the class schema. Perhaps the most well known of these is Object-Z [41, 63].

The class schema enables the grouping of operations to act on a particular state schema existing inside the class. A class schema typically contains:

- A *visibility list* to restrict access to the listed features in objects of the class. A feature may be one or more constants, state variables or operation schemas.
- *Type and constant definitions* as found in Z.
- *Inherited classes* to enable re-use of class schemas through inheritance.
- A *state schema* given as a nameless schema that contains state variables with associated predicates.
- An *initial state schema* distinguished by the keyword *Init*. This defines possible initial states for instances of the class.
- *Operation schemas* to model the behaviour of the class. The operation schemas are different to those found in Z in that they have a Δ-list of state variables whose values may change when the operation occurs. In Z, Δ normally applies to entire schema types.
- A *history invariant* which acts as a predicate over histories of objects of the class which further constrain the behaviour.

Object-Z also introduces the parallel operator $\|$ to enable inter-object communication. This operator conjoins operation schemas and matches output variables with input variables. These variables are then hidden. Detailed case studies using Object-Z have been developed. Examples of these include [61, 62].

Object-Z is a satisfactory specification technique for modelling isolated objects or simplistic interactions between objects. For modelling more complex (distributed) systems the technique is limited. There are several reasons for this. Firstly, Object-Z does not separate syntactic issues from behavioural issues. As a result, issues in the syntactical aspects of composition are not addressed. Similarly, satisfaction of any operation naming rules associated with a class schema, such as those prescribed by ODP [109], are not readily enforceable by the language, *i.e.* they require the specifier to follow a naming policy as opposed to prescribing the policy. The language is also limited in the forms of composition it offers. Ideally a language would provide a multitude of behavioural composition possibilities, *e.g.* as in LOTOS (see section 3.3.3).

Other extensions to Z include MooZ [166], OOZE [2], Schuman & Pitt [169], Z++ [122] and ZEST [55]. The last of these is worth mentioning in more detail since it was designed specifically to be suitable for the specification of distributed systems. Also, it does not attempt to modify the semantics of Z. Rather the class schema it provides is merely a syntactic construct that can be flattened to produce a standard Z specification.

A class type in ZEST captures the common behaviour pattern of a set of objects — its instances. We present here the original description of a class as given in [55]. The work on developing ZEST is on-going, however, and more recent extensions [56] include features such as visibility lists as found in Object-Z. This work is not yet stable, so we restrict ourselves here to the original version of ZEST. Classes are represented in ZEST by schemas of the following form.

TYPE

Attributes
States
Operations

Here *Attributes* are an axiomatic description. They introduce attribute variables and the relationships between them. These variables have values that are fixed for an object of that class. *States* is a schema. It declares a number of state variables and the relationships between them and the attribute variables. The values of these variables represent the current state of the object and they may be changed by operations. *Operations* denotes a set of named schemas. Each schema declares ways in which state or attribute variables can be accessed and the effect the schema has on the values of these variables. The *Operations* set may

contain an *Init* schema which declares the valid initial states for an instance of the class. If omitted, then any state may be an initial state.

The above description is similar to Object-Z in many respects. ZEST, however, provides a formal theory [52, 53, 54] for reasoning about inheritance and also perhaps more importantly, given the discussion in 2.1.1, subtyping. This work provides a formal representation of what it means for one object to be able to replace another object. This was achieved through the idea of *extension*. Briefly one class *s extends* another class *t* in ZEST when:

- the attribute and state space of *s* can be projected in a natural way onto the attribute and state space of *t*. That is, every attribute and state variable of *t* is also represented by a corresponding attribute or state variable respectively in *s*. The value *v* of every common attribute or state variable valid in *s* should also be valid in *t*.

- to each operation P_t in *t* there is a corresponding operation P_s in *s* such that P_t has a weaker precondition and stronger postcondition.

Whilst an elegant theory was developed for subtyping, this was found to be limited in its applicability compared to other current approaches to specifying behavioural type relations. This work provided a theoretical basis for reasoning about such issues, but as with the action oriented specification approaches, this was not directly available in the specification.

Other Approaches to Specifying Objects in Z

The previous section gave a brief overview of the numerous extensions to Z that have been proposed to specify systems of objects. As stated in section 3.3.5, Z allows for the specification of objects but does not strictly support classes and inheritance.

There exists ways in which Z can be used, however, in a more object-based style without the need for extensions. We consider the two main proposals for specifying object systems in Z: Hall's approach [91, 93] and ZERO (Z Expression of Refinable Objects) [209, 210], as developed by Whysall and McDermid.

In Hall's work conventions in specification style are used for modelling object-oriented systems. Brownbridge [37] describes a substantial implementation where this style was successfully used. There are five main ideas on which Hall's style is based:

- conventions for modelling object states;

- use of object identities to refer to objects and express their individuality;

- a convention for expressing the state of the system in terms of the objects it contains;

- use of object identities to model relationships between objects;

- a method of defining operations in terms of single objects and calculating their effect on the whole system, or on defined sets of objects.

Hall in a later paper [91] goes on to show how classes and inheritance between classes can be modelled in Z. This can be achieved provided an extensional notion of class is used as opposed to an intensional one.

In ZERO, objects are described by so-called export and body specifications. Export specifications are algebraic in style and describe the overall behaviour of the object independent of the internal details of the objects. Body specifications are model oriented descriptions of the constituent parts of each objects, in particular their state and methods. Body specifications are typically given in standard Z. The body specifications are subsequently used as the basis for refinement of the objects. Proof obligations are required to ensure that the body and export specifications actually represent the same behaviours.

3.4 Tool Support for LOTOS and Z

Tool support for FSLs is critical both to their successful usage and to their uptake generally [50]. Tool support can take many forms. We provide an overview of the different sorts of tools available for LOTOS and Z. As well as being of relevance in its own right, this overview also highlights the semantic approaches that underlie the two languages and shows their orthogonality.

3.4.1 Tool Support for Z

Tools to support Z are becoming more available and steadily being taken up by the Z community. A summary of Z tools can be found in [157] and [185]. It is argued in [7], however, that tool support is not necessarily the answer to producing good formal specifications. Tools are not a substitute for thinking, and producing specifications is essentially a cerebral activity. More often than not, the best specifications are drafted with pencil and paper and without recourse to a tool in the initial stages. Tools are important in the later stages of specification development, however, especially when the specification becomes large and navigation through the specification is non-trivial. Tool support is also useful for identification of minor errors in the specification, *e.g.* type errors.

Z tools offer various options such as: type setting, syntactic checking, cross referencing and navigation, schema expansion, type checking, theorem checking and proof assistance, animation and code generation.

There does not exist presently any single tool that possesses all of these options. FuZZ [182] for example, provides only syntactic checking and type checking. The generation of the Z formatting requires a detailed knowledge of the *LaTeX* processing environment. Other tools such as Formaliser [74] offer a more complete environment for production of Z specifications. Cross referencing and navigation are provided as well as a user interface to access the Z symbols. CaDiZ [193] is one other well developed toolset that contains several of the above features.

Tool support for theorem proving is still, by and large, a research topic. Recent work suggests that a certain degree of proof is possible. Machines will never be able to prove every theorem, but there is no point in reproducing a proof that has been done many times previously and can be regurgitated by a machine. One such theorem proving assistant is **J**igsa$_w$ [154]. This has incorporated the deductive system of standard Z. **J**igsa$_w$ is supported by the tactic language *Angel*. This allows proofs to be defined in a general and reusable way. Through providing a basic minimum of deductive rules, tactics for complete proofs can be developed. An introduction to the Gentzen-style sequent calculus that forms the basis for the deductive system of Z as given in the Z standard [106] can be found in [180].

Animation and code generation are also very much in their infancy. Tool support for code generation as found in [199] and [140] have shown that a restricted subset of Z can be implemented. The extension to Z in its entirety is highly unlikely however due to its expressiveness. Z can be used to specify systems that can never be implemented. A more general approach to developing an executable semantics for Z is given in [34].

3.4.2 Tool Support for LOTOS

Tool support for LOTOS is much more developed than that currently existing for Z. Tools are available for dealing with the syntactic aspects of specifications. Examples of these tools include:

- syntax directed editors such as the Cornell Synthesizer Generator editor [200] and the graphical editor for G-LOTOS [42];
- syntax editors such as SCLOTOS (Syntax Checker for LOTOS);
- cross-referencers such as LXREF (LOTOS Cross Referencer);
- report generators such as REPADT and REPDEP;
- report browsers such as LOBROW (LOTOS Browser);
- graphical browsers such as G-LOTOS (Graphical LOTOS);

- static semantic analysers such as LISA (LOTOS Integrated Static Analyser) and TOPO (LOTOS Compiler) [134].

LOTOS also has many tools that have been developed for analysing and verifying properties of specifications. Examples of these tools include:

- AUTO [132] for analysing and manipulating labelled transition systems;
- COOPER [1] for deriving canonical testers based on the COOP method [207].
- PERLON [21] for checking the persistency properties of ACT ONE data types.
- SQUIGGLES [7] [24] for automatically checking behavioural equivalences of LOTOS specifications.
- SMILE [64] for amongst other things enabling the symbolic simulation of LOTOS specifications.

LOTOS also has numerous tools that allow specifications to be translated to implementation languages. Examples of these include TOPO [134] and COLOS (LOTOS to C Compiler).

There now exist several toolsets that encompass many of these individual tools so as to provide a complete LOTOS environment for developing, verifying and finally implementing LOTOS specifications. Examples of these toolsets include the LOTOSPHERE Integrated Tool Environment (LITE) [201] and the CAESAR/Aldebaran toolset (CADP) [69]. LITE was developed through a European collaborative project. CADP was developed jointly by two French companies.

3.5 Conclusions on Formal Languages for Distributed Systems

The previous sections have given a broad summary of the formal language family. Five sub-families of the specification language spectrum were considered. Each of these has its own immediate advantages and disadvantages for describing aspects of distributed systems. These summarise these advantages and disadvantages in the following sections.

3.5.1 Conclusions on Equational Specification Languages

Equational specification languages offer a means to represent directly the syntactic aspects of potentially complex behaviours in a checkable manner, *i.e.* they support directly the construction of abstract data types. One problem with algebraic specification languages though is that they themselves do not allow this checking to be carried out in a flexible enough manner. That is, they use name comparison only as a basis for type checking. Structural relationships, *e.g.* isomorphism, between sorts is not used when type checking is done. As a result, they do not in themselves support inclusion polymorphism. A further problem with algebraic specification languages is that the equations associated with them, *i.e.* the axioms (behaviour) that the operations are expected to satisfy, are typically verbose and difficult to write correctly.

3.5.2 Conclusions on State Oriented Specification Languages

State based approaches offer certain advantages in describing object-oriented distributed systems. State oriented languages allow reasoning about the possible states that processes (systems) can get into and in turn the states that cannot be reached. It might be argued that this feature goes against the fundamental object oriented principle of encapsulation. It is argued in [33] that certain states that dictate the visible behaviour, *i.e.* the behaviour of an interface for example, do not necessarily have to violate encapsulation. For example, an automated teller machine might have control states: *wait for card, wait for number, give money*. These control states do not as such violate any principles of encapsulation. Rather, they are another means of abstractly representing potentially complex behaviours that might arise in an interface. Encapsulation it is argued would be violated when states were made visible that were only used for specific internal action sequences.

[7] So called because of the symbols used to describe strong and weak observational equivalence, *i.e.* $\approx$ and $\sim$.

3.5.3 Conclusions on Action Oriented Specification Languages

Action based approaches offer several direct advantages for specifying and reasoning about distributed systems. They support modularity directly through processes. These processes allow complex behaviours to be specified through action orderings. Further, there exist numerous ways in which these processes can be combined to form interacting systems. This idea of sub-systems that interact with one another is well matched with models of distributed systems. As well as these features, process algebras have a well defined semantic basis that has been used to develop detailed theories on behavioural relationships between systems.

Process algebras are not without their limitations though. In distributed systems, type checking is used as a basis for deciding whether operations on remote interfaces are possible. This is normally done on a syntactic level. As argued in section 3.2, formal languages should support this syntactic checking as well as augmenting it with other aspects. This is not possible to do directly in process algebras. It is possible to use Act One in such a way that a contrived form of syntactic type checking can be achieved. This is presented in chapter 7. Aspects that relate to non-functional requirements are not well suited by process algebras though. There exists much current research and standardisation activities [153, 112] that are investigating how process algebras can be extended to deal with non-functional issues of systems.

Another limitation of process algebras for specifying distributed systems is that whilst elegant theories have been developed that relate process behaviours, these theories are not usable in the specification directly. For example, it is not possible to reason within LOTOS, about whether two processes that synchronise with each other will have some desirable or undesirable behaviour. Such considerations are normally considered outside the specification when testing or reasoning about the specification is done. Whilst LOTOS relations such as extension appear to deal directly with behavioural type issues, *e.g.* there is a close relationship between extension and inclusion polymorphism, such relations are not directly accessible in the specification.

3.5.4 Conclusions on Logic Based Specification Languages

Logic based approaches allow for a higher level of abstraction when specifying. They enable reasoning about the system in terms of assertions, that is, predicates that may evaluate to either true or false. This property is a particularly useful feature in formal specification languages for describing distributed systems. For example, consider non-functional assertions, *e.g.* statements like the cost of using a service should be less than a certain number, can be specified directly. Modal logics in particular offer a powerful means to reason about about aspects of the behaviour of systems, *e.g.* once invoked some operation will return a result within a certain time frame.

Whilst being very powerful, logic oriented languages on their own are limited as a specification approach due to the lack of modelling and structuring concepts they offer. Often, other approaches are used to specify systems and logic based approaches are used to reason abstractly about these specifications. Examples of this approach include [16], where LOTOS is used to describe the behaviour of systems and a temporal logic (QTL) is used to impose real-time constraints on the system. This dual language approach is argued to allow for a separation of concerns: the functional behaviour and real-time issues.

3.5.5 Conclusions on Model Based Specification Languages

Model based approaches in their classical usage offer a means to describe abstractly, very complex systems. For example, the ability to write down global predicates that the system as a whole must satisfy allows a powerful specification approach. This abstraction mechanism may in some respects be regarded as the main advantage as well as disadvantage of model based approaches. Whilst it is quite possible to make such global assertions and statements, this high-level of abstraction means that the step from specification to the final implementation is made much larger than in other more constructive FSLs. Refinement approaches that relate abstract specifications to less abstract ones have been developed to address these issues. Typically, these require a proof to be made that relates a concrete to an abstract specification, *e.g.* the preconditions and postconditions for an abstract operation should be satisfied by the refined operation. Despite such approaches, the complexity involved in realising, *i.e.* implementing, completely abstract specifications of

complex systems remains vast due to the amount of proofs that have to be carried out in the refinement process.

With regard to distributed systems in particular, model based approaches are also limited in their classical usage, since they do not address issues such as concurrency or non-functional behaviour. Further, their support for modularity, *e.g.* objects, is limited.

3.6 Summary

This chapter has attempted to provide some insight into formal languages generally and give insight into the particular requirements on languages used to describe distributed systems. Ideally, we would like a language that supported :

- the syntactic aspects of types as found in algebraic specification languages;
- the behavioural properties and reasoning facilities as found in action oriented languages;
- the ability to reason about states generally as found in state oriented languages;
- the ability to express concisely properties that the system should have as found in logic oriented languages.

In the following chapters we shall see how LOTOS and Z can be applied in such a way that they address many of these issues.

Chapter 4

Development of an Architectural Semantics

This chapter argues in detail that an architectural semantics can alleviate many of the problems involved in developing specifications for a particular problem domain. We show how LOTOS and Z may be used to develop an architectural semantics for distributed systems based upon the reference architecture of Open Distributed Processing. The advantages and disadvantages of these techniques are discussed with respect to the concepts and structuring rules they support and those that they cannot directly support.

4.1 Architectural Semantics

It is often the case that writing specifications proves to be difficult due to poor initial choice of specification structures. Thus having a good architecture upon which specifications can be based removes many of the difficulties involved in the actual writing of specifications. By a similar argument, specifications written without a well structured architecture tend to be not only difficult to write but also hard to understand and difficult to modify and extend.

Having a good specification architecture is also very useful for problems that are not well defined by requiring detailed consideration of the informal problem statements. Thus attempting to formalise "messy" problems directly can lead to "messy" specifications.

An architectural semantics[1] involves taking an informally defined reference architecture and interpreting it in a formal language. We shall investigate different approaches to this interpretation shortly. The theory behind the development of an architectural semantics is that by interpreting the most basic of concepts underlying a given architecture, specifications can be engineered. That is, provided that the architecture itself introduces concepts in a hierarchical fashion,[2] specifications can be structured around a base set of formalised concepts. >From a specifier's viewpoint this has several advantages. It corresponds to specification re-use — the holy grail of software engineering from the specifier's viewpoint. An analogy here would be an electronic engineer who works at an architectural level. The engineer does not have to re-specify the most basic of components such as flip-flops and NAND gates, but rather may use these as building blocks to create more complex components. An approach using LOTOS to do exactly this may be found in [174] and [173].

As well as the possible advantages for specifiers, the development of an architectural semantics has numerous other advantages. An architectural semantics provides clear and concise statements in a given formal language — a formalisation of concepts which then acts as a more precise definition of the informally defined architectural concept. Doing so requires a more in-depth consideration of the textual definition of each concept than might otherwise have been achieved. Thus "intuitively clear" concepts which might be open to different interpretations are made more precise and any ambiguities are removed. We shall shortly

[1] This term was first used by Prof. Chris Vissers, University of Twente.

[2] We shall see that this is the case with the architecture adopted in this thesis: the ODP-RM.

see examples of the problems of informality in developing specifications for OSI.

In defining an architectural semantics, the developers of the architecture itself may have confidence in their architecture if it can be interpreted in a formal language. The architectural semantics acts as a bridge between the concepts of a given architecture and the semantic model of a given formal specification language. Concepts that cannot be interpreted in a given formal specification language need not necessarily be wrong. It might simply mean that this concept is not well matched by the semantic model of the given formal specification language.

An architectural semantics also offers the basis for comparison of different formal specification languages when used to provide formal descriptions of the same standard. Hence it also helps in identifying which formal specification language is most suitable for a given problem domain.

Notions such as conformance, consistency and compliance may also be addressed through the development of an architectural semantics. Advantage can be taken of existing tool support for formal languages that have been used to develop an architectural semantics for the architecture under consideration.

There are several indirect advantages that arise out of the development of an architectural semantics. Perhaps the most important of these is in clearing up the textual description of the architecture under consideration. This of course should ideally be done in conjunction with the development of the architecture itself. This was the case with ODP but not the case with OSI. In the case of ODP numerous ambiguities and statements that were unclear or misleading were identified and subsequently rectified. We shall see that despite the architectural semantics work, several problems with the reference model of ODP still exist. We discuss these in the rest of this chapter and the following chapter.

By developing an architectural semantics the limitations of the formal specification languages used are also identified and documented. These can then be used by formal specification language developers to extend and improve existing formal specification languages.

It should be pointed out that the development of an architectural semantics should not be focused on showing that two arbitrary specifications written in different formal languages are the same (though, of course, the equivalences defined for the specification languages should help here). An architectural semantics is also not about redefining architectural concepts in a form more suitable for formal languages, or adding/removing architectural concepts that can/cannot be interpreted in given formal languages. The development of an architectural semantics might, however, result in the last of these by making the architecture developers reconsider the existing concepts.

An architectural semantics can be used by anybody interested in that particular architecture. These may include:

- developers of the architecture themselves;
- developers of standards to be generated from that architecture;
- implementors whose products comply with standards generated from that architecture;
- testers of conformance to standards generated from that architecture;
- end-users of products designed according to standards generated from that architecture.

The idea behind development of an architectural semantics is not new. Indeed the architectural semantics work of ODP originated from problems with OSI. We discuss these problems briefly in the following subsection.

4.1.1 Historical Background

Formal specifications of OSI standards gave scope for different interpretations of architectural concepts. This was not in itself wrong, but simply reflected the generality of the architecture. Interpreting informal concepts in formal specification languages requires attention to how a given concept should be understood. Some of the problems identified included:

- Service primitives in OSI model interactions at services. Service primitives were not defined in the OSI Reference Model (OSI-RM) [103] but in the OSI service conventions [102]. It was not stated

whether service primitives were atomic, instantaneous or synchronous. Thus specifiers could regard service primitives as procedure calls or asynchronous requests (in SDL [118] and ESTELLE [100]) or synchronous calls (LOTOS [101]). This was not just hair-splitting but led to radically different behaviours being specified, *i.e.* different implementations of the same standard.

- Service data units had different interpretations. It was not clear whether they were atomic or not. Hence different behaviours were possible, *e.g.* protocol data units could be sent off before a given service data unit was completely received by a protocol entity.

- Service access points had different interpretations:

  * Did they reflect a structural concept, *e.g.* an interface between two protocol entities?
  * Were they active agents, *e.g.* did they have a dynamic aspect through which connections could be established?
  * Could they be represented by processes which could be further decomposed?
  * Could connection-less and connection-mode services be supported at the same service access point?
  * Were endpoints necessarily associated with connections or were they a more general concept?

A fuller account of the historical reasons for the development of an architectural semantics for OSI may be found in [195].

4.1.2 Realisation

The first questions that arise when considering the development of an architectural semantics are: where does the architecture come from? As discussed in section 2.3, we focus in this thesis on the architecture provided by the ODP-RM. In particular, we choose the basic modelling and specification concepts of the ODP-RM, and certain of the ODP viewpoint languages.

The next immediate question that arises is: what formal languages should be used to develop an architectural semantics? In principle any formal language could be used. However, to be particularly apposite the formal language should have a solid semantics and be well known. There should also exist a source of expertise in that particular formal language, since developing an architectural semantics often requires specification choices to be made. Insight into the consequences of selecting a particular way of representing a given concept, and the ramifications of that choice, should be seen as far ahead as possible. Further, using the language to specify concepts and structures that it was not specifically developed for requires detailed knowledge of the limitations and application of the language. We shall see how, in developing an architectural semantics for ODP, the support for certain concepts and structuring rules did not exist directly. However, the language could be used in a certain stylised way to achieve the desired effect.

The formal languages used to develop an architectural semantics for ODP included: LOTOS [101], SDL'92 [118], Z [106], and ESTELLE [100]. All of these formal languages have their own particular advantages and disadvantages in formalising the architecture of ODP. We focus in this thesis on LOTOS and Z, which as discussed in section 3.3 represent rather different specification languages.

For any given formal language, there are three main ways that it can be used to formalise a given architectural concept:

- an approach based on interpretation;

- an approach based on providing specification templates;

- an approach based on mapping from a direct (mathematical) interpretation of that concept.

Each of these approaches has both advantages and disadvantages which will now be discussed.

Interpretation based Approach

This approach interprets how a given architectural concept might be represented in a given formal language. The result of this is a precise natural language statement offering how that concept might best be represented in that formal language. It is quite possible, and as we shall see it is the case, that choices exist as to how concepts can be represented in a given language. The interpretation based approach should document these choices and, where possible, give advice on the advantages and disadvantages of modelling concepts a given way.

The advantage of an approach based upon interpretation is that it enables an in depth comparison of all concepts in all formal languages. Through this approach the semantics of all of the concepts may be checked against the semantic models of the formal languages. In doing so, more understanding of the concepts is developed. The approach also gives specifiers guidance without being prescriptive as to how they should specify certain concepts. This last point is also perhaps the greatest weakness of the interpretation based approach. For instance, as it not prescriptive it is not possible to identify immediately whether any given specification is compliant with that architecture. Similarly, the lack of prescriptivity prohibits specifications from being built directly. Instead, explanations as to how the concepts and structuring rules are to be represented provide the only information for specifiers. Whilst useful in itself, a more directly accessible approach is to provide specification templates.

Template based Approach

This approach is based on providing specification templates for concepts and structuring rules. Through this approach, a structuring of specifications can be achieved. This approach closely corresponds to our analogy given earlier of the electronic engineer. Unfortunately, as we shall see, it is not the case that such an approach is always possible. Certain concepts are so generic, *e.g.* type as presented in section 4.3.7, that developing specification fragments for them is impossible. Further, certain concepts cannot be modelled directly in specification languages, *e.g.* subtype as presented in section 4.3.9. Another limitation with a template based approach is that it does not allow arbitrary specifications to be checked for some form of compliance to the architecture under consideration.

Mapping from a Direct Mathematical Interpretation

The theory behind this approach is based on giving a direct mathematical interpretation of the concepts and structuring rules forming the architecture under investigation. This mathematical interpretation might, as in the case of the work done on ODP [148], take the form of transition rules which characterise valid behaviours of systems of interacting objects. >From this mathematical interpretation, mappings to different formal languages can be defined. It is argued in [148] that such an approach allows the semantics of architectural concepts to be defined directly in mathematics, as opposed to mapping concepts onto the (syntactic) features of specification languages. As a result, a basis for language comparison can be made formally through providing mappings to specification languages. Unfortunately, the reality of developing such an approach for ODP is somewhat different. Mappings to different specification languages have not proven possible for two main reasons. Firstly, the fundamentally different semantic bases underlying specification languages. Trying to find a common bases for comparison of different formal languages although perhaps possible in certain cases, *e.g.* [211], is unlikely to exist in the main. One of the main reasons for this specification equivalent of the grand unified theory of physics [206], is that specification languages are often used for entirely different reasons. From the specification of low level intricate behaviours as might be found in a communications protocol to broad statements that relate to entire systems and enterprises. Secondly, and perhaps more problematically, mappings have not proven possible due to the lack of prescription given in the ODP-RM. As stated, this is a natural consequence of the role of ODP. It is a standard for developing numerous distributed systems which may have completely different properties. Hence there is a limit to how prescriptive the architecture can be. This issue and the problems it causes in developing an architectural semantics are considered in detail in the rest of this and subsequent chapters.

There are other problems with this approach. Perhaps the greatest drawback is that it is very (overly?) mathematical. Formal methods often offer a symbolic meta-language which to a great extent hides their

mathematical foundations. With this direct mathematical formalisation, the (mathematical) foundations are blatantly visible and hence not as accessible to people with a non-formal background.

This approach by itself also does not offer any means to develop specifications. It simply represents the generic mathematical interpretation of a subset of the architecture under consideration. In effect the direct formalisation done for ODP [148] represented a part of the computational language, specifically, the operational interactions of the computational language. Extending the work to cover more of ODP would be problematic due to the lack of prescription in the concepts that exist there.

Another issue with this approach is the question as to whether the direct formalisation correctly represents the architecture in the first place. Since the architecture is itself informal, formalisation of it can only ever be done informally. Formality exists once a model is complete and reasoning about the model is done formally, *e.g.* proving properties about the modelled system. Indeed several technical problems have been found in the direct formalisation of [148] as it currently exists.

4.1.3 Conclusion on Approaches

Each of these approaches has its advantages and disadvantages. The best possible approach that could be taken in developing an architectural semantics for a given architecture would consist of:

- providing templates in all formal languages for all of the concepts contained within the architecture and their combinators;
- being able to check arbitrary specifications for compliance to that architecture.
- being able to check when specifications written in different languages for that architecture are in some sense compatible.

We shall see in the rest of this chapter and the following chapter just how far the development of an architectural semantics for ODP has realised this ideal. Before doing this, it is useful to note the distinction between the terms “architectural semantics” and “specification architecture”. A specification architecture shows how a specification is structured. An example of an approach to develop specification architectures is given by Vogel’s work [203]. His approach was to structure LOTOS specifications, or more specifically the process definitions in the process algebra part of a LOTOS specification, in a hierarchy that represented the hierarchical structuring of the concepts in ODP. Unfortunately, due to the necessary lack of prescription in ODP his work fell short in that the processes had little or no behaviour or information associated with them. Instead his work provided a syntactic structuring based on process names that corresponded to concepts in ODP. Whilst not altogether a bad starting point to writing specifications the work had problems in that it provided little in the way of documented limitations in the approach. It could also be argued that the work was overly prescriptive in the way the concepts were presented.

An architectural semantics on the other hand is wider in scope. Of course, ideally the architectural semantics should result in the production of specification architectures, we shall see how true this is in the rest of this chapter and subsequent chapters. The development of an architectural semantics should also provide detailed analysis of both the architecture under investigation (here ODP), the limitations of formal languages and outline suitable improvements and extensions to those languages.

4.2 Formalising the Basic Modelling Concepts

In developing a variety of architectures of similar broad type, certain concepts repeatedly arise. These concepts reflect the domain under consideration directly. That is, they model aspects of the real world in which our domain exists. We, like ODP, term these basic modelling concepts. Given the advantages offered by object technology with regard to distributed systems and discussed in section 2.2, the basic modelling concepts given in ODP and formalised here are concerned with objects and the concepts associated with them.

We attempt to introduce these concepts in a bottom up fashion. As will be seen this is not always possible due to their close inter-relationships. We shall see that in certain cases we offer two formalisations in Z which we term **Classic Z** and **Abstract Z**. The first of these corresponds to how concepts may be

represented in Z in its classical usage of state and operations. The second of these corresponds to using the language in a more abstract manner to formalise the concepts. As will be seen in the following chapter, the latter approach offers a great deal more flexibility with regard to reasoning about concepts and, importantly, the structuring rules that apply to them.

It should be pointed out that we restrict ourselves in our abstract Z approach to those concepts needed for the following chapter. Specifically we give those concepts related to the computational viewpoint of ODP, as it is here that our work focuses in detail. The primary reason for our focus on this viewpoint language and not the others is that ODP is more prescriptive here. Hence it offers us more potential to develop specification architectures. We note also here that formalisations without a label, *i.e.* classic or abstract, are to be taken as Z in its classic usage.

It should also be noted that LOTOS in effect offers two languages: the process algebra and Act One. Our formalisations here focus on the process algebra part. It is quite possible to formalise many of the concepts in Act One also [110]. Whilst offering certain advantages in some cases, this approach has not been presented in detail here. We shall see in the following chapter how some of these concepts may be formalised when we consider aspects of the information viewpoint formalisation.

The formalised concepts are introduced as follows. Firstly an informal discussion relating to the concept being modelled is given. This is loosely based on the text of the ODP-RM [108]. Following this we show how the concept may be represented in LOTOS and then Z, both in its classical usage and in a more abstract manner.

4.2.1 Action

Architecturally, the most fundamental of all concepts is arguably the notion of an action. An action is something that happens. In the real world, actions are typically at arbitrary levels of abstraction. The action of closing a door for example could be regarded as a single action. At another level of abstraction though, it may well be represented by a collection of actions such as approach door, touch door, push door for example. From a modelling perspective, this distinction is critical since it reflects directly on our specifications, *e.g.* actions may overlap in time and they need not be instantaneous.

Every action of interest for modelling purposes is associated with at least one object. Actions themselves can be partitioned into two groups: **internal actions** and **interactions**. Internal actions occur without environmental participation. We define environment in section 4.2.6. Interactions take place with environmental participation. This latter case represents observability of the action. Internal actions are invisible to the environment of the object. It is often useful to consider interactions in terms of cause and effect relationships between the interacting objects. This is especially so if the client/server paradigm is used.

Formalisation in LOTOS

Actions in the process algebra part of LOTOS are modelled as either internal events or observable events. All process algebra events are atomic. An internal action may be given explicitly by the internal event symbol, **i**, or by an event occurrence whose associated gate is hidden from the environment.

An interaction is represented by a synchronisation between two or more behaviour expressions associated with objects at a common interaction point (gate). Interactions may be of the kind:

- pure synchronisation on a common gate with no offer: no passing of values between objects occurs;
- **!** and **!** for pure synchronisation: no values are exchanged between the objects;
- **!** and **?** for value passing provided the **?** event contains the **!** event: another way of considering this is that the **!** event selects a value from a choice of values for the **?** event;
- **?** and **?** for value establishment: here the effect is an agreement on a value from the intersection of the set of values. If the intersection of the values is the empty set then no synchronisation and hence no interaction occurs.

If a non-atomic granularity of actions is required, event refinement may be used. This will then enable non-instantaneous and overlapping actions to be modelled. It should be noted that event refinement is a non-trivial problem, especially when behavioural compatibility is to be maintained. The issues surrounding event refinement in process algebras are considered in detail in [49].

There exists no construct in the process algebra part of LOTOS to express cause and effect relationships, although this might sometimes be possible to represent informally. Value passing may very informally be regarded as the sending of a value from one process to another which has certain flavourings of client/server causalities, *i.e.* one process sends a message to another process to perform some service. Semantically, this causality is not explicit. Process synchronise instantaneously or not at all. There is no real notion of sending a message between objects. This is further exemplified by synchronisations involving multi-way value passing, *e.g.* event offers with action denotations such as *g ?x: Nat !true;*.

It might also be the case that an Act One sort can be used to label an event offer with a given causality. This choice of course requires that the equality or inequality of the values of these sorts can be checked.

Formalisation in Z

Classic Z: An action is modelled in Z by the performance of an operation specified in an operation schema. The effect is the instantaneous change in state (or the null change) of the objects with which that action is associated. An action may produce a non-deterministic result.

Since there is no explicit notation of encapsulation in Z, it is not usual to determine whether an action is observable or internal in Z, hence the distinction between interactions and internal actions is not clearly defined. One convention that can be used to make such a distinction is through the identification of inputs, outputs or global variables in operation schemas. Operation schemas requiring inputs, producing outputs or accessing global variables may be regarded as observable and hence must interact with their environment. The environment itself may or may not be specified (see 4.2.6). Actions requiring inputs from an unspecified environment that produce no outputs may be regarded as externally invoked non-observable actions. Actions producing outputs going to an unspecified environment may be regarded as internally invoked (spontaneous) observable actions. Actions that require inputs from an unspecified environment and produce outputs going to that environment may be regarded as externally invoked observable actions.

If the environment of an object is specified, however, then this implies that for each operation schema requiring inputs or outputs that is associated with an interface to an object, *i.e.* an observable action, there exists another operation schema (possibly associated with another object) that requires inputs or outputs of the same type as the object under consideration. These two operation schemas are then conjoined with the inputs/outputs of the operation under consideration being renamed as the outputs/inputs of the operation representing the environment respectively.

Alternatively the occurrence of operations that reference variables that are global throughout the specification can be regarded as interactions.

All operations in Z are atomic. That is, operation schemas in Z either happen in their entirety or do not happen at all. Thus it is not possible in Z to have actions that are not atomic. This includes actions that consist of several other actions. For example, if two actions are combined through the schema calculus composition operator ($\stackrel{0}{9}$) to form another action, then the occurrence of this action composite action is not one action followed by another. Rather the two actions occur together or not at all, *i.e.* it is not the case that a third action might occur elsewhere in the specification between the occurrence of the first and second actions of the composite action under consideration.

The notion of cause and effect relationships are not strictly within the scope of Z. However, if an operation requires an input to occur, then this might be considered as the environment causing this operation to occur, *i.e.* the environment acts as the producer and the operation schema as the consumer. Similarly, if an operation schema produces an output then this might be considered as the environment acting as the consumer and the operation as the producer. If a given operation schema requires inputs and also gives outputs, or has no inputs or outputs, then it is not possible to give a cause and effect relationship to that particular action.

It should be noted that the syntactic convention for distinguishing internal and observable actions is limited since there is no semantic distinction between operations which are to be interpreted as spontaneous or internal, and those which require environmental participation; this is left for the natural language

commentary which should accompany all Z specifications. As a consequence of this, the above definition treats a lossy queue as a subtype of a queue. Clearly though the intention of a lose operation in a lossy queue is that the operation should occur non-deterministically.

Abstract Z: Z can be used to consider actions at different levels of abstraction. The highest level of abstraction at which an action could be considered in Z would be to introduce actions as a basic data type. This approach could then be used to denote actions without being particularly interested in what they do.

Typically, we are interested in the consequences of performing actions. In classical Z, actions are represented by operation schemas relating states. At a more abstract level an action is simply a relation. Actions have a syntactic element to them, *e.g.* their name and the possibly empty set of parameters they accept as input or return as output. The domain of the relation represents the state before the occurrence of the action. The range of the relation represents the state following the occurrence of the action. Given that actions can be either observable or internal, an abstract representation of an action as represented by a relation might be:

$$Action ::= OAct\langle ObsActSig \times State \times State \rangle \mid IAct\langle IntActSig \times State \times State \rangle$$

Here *Action* represents the actions under consideration; *ObsActSig* and *IntActSig* represent the signatures of the actions (see section 4.3.11). *State* represents the states that actions relate. *OAct* and *IAct* represent the relations between signatures and states that produce a model of actions. This model allows the syntactic aspects of actions to be reasoned about. Actions modelled this way cannot be used to represent cause and effect relationships. This can only be done informally through the inclusion of an identifier either in the domain of the action relation or in some context where the action exists, *e.g.* in the interface if the action is associated with the interface behaviour.

It should be noted here that this modelling of internal and observable actions does not take into account the spontaneity of internal actions.

4.2.2 Behaviour (of an object)

Behaviour generally is something that happens. Typically it consists of a collection of actions. At a higher level of abstraction, a behaviour can correspond to a single action. The actions in a behaviour have constraints on their occurrence. These constraints may be expressed directly, *e.g.* through some form of action ordering, or indirectly, *e.g.* through state information or environmental participation. The specification language in use determines the constraints that may be expressed.

Formalisation in LOTOS

The behaviour of an object is defined by the LOTOS behaviour expression associated with the process definition that constitutes the object template. A behaviour expression may consist of externally visible event offers and internal events. The actual behaviour of an object as might be recorded in a trace, is dependent upon the behaviour expression associated with the object and how this is composed with the environment. The actual behaviour the object exhibits depends upon the behaviour expression of the object and how this synchronises with its environment. An object may exhibit non-deterministic behaviour.

Formalisation in Z

Classic Z: The behaviour of an object in a given state is the set of all possible activities that may occur from that state. The actual sequence of actions that may occur may be affected by the environment of the object and the constraints expressed in the preconditions.

Abstract Z: an abstract model of behaviour, where the actions are relations between states, is simply a relation. Two actions are in the relation if the state following the occurrence of the first action is such that it makes the occurrence of the second action possible. This may be represented as:

$$behspec == \{ar : action \leftrightarrow action \mid (\forall as1; as2 : ActSig;\ s1; s2; s3; s4 : State \mid (Act(as1; s1; s2); Act(as2; s3; s4)) \in ar \bullet s3 \subseteq s2)\}$$

Here *ActSig* corresponds to the signature of the action that the *Act* relation uses in relating before and after states. We model the state satisfaction relation through subsetting of states. It should be noted here that there is a distinction between behaviour occurrence and behaviour specification. Similarly, there is a difference between action and action occurrence. The action occurrences in a given behaviour are always distinct. The action templates in a behaviour specification need not necessarily be distinct though. As a result, this model of a behaviour specification as a relation is best suited to dealing with orderings of distinct action templates. We discuss this issue of action template and action occurrence again in section 4.2.5.

To overcome the problem of repeated action templates in a behaviour specification, it is possible to use bags in Z. A bag may be regarded as a set that allows for repeated membership. Thus another model of behaviour that allows for repeated action templates is:

$$bs : \text{bag}\ (action \times action)$$

$$\forall as1; as2 : ActSig;\ s1; s2; s3; s4 : State \mid (Act(as1; s1; s2); Act(as2; s3; s4)) \sqsubseteq bs \bullet s3 \subseteq s2$$

Here the operation $\sqsubseteq$ checks whether an element is a member of a bag or not.

Other models of behaviour are also possible in Z. For example, a behaviour might simply be regarded as an action ordering without consideration of the states that actions relate. Consider the LOTOS behaviour *a;b;c;*. This might be represented in Z by a relation containing the two pairs *(a,b),(b,c).* Typically such relations form a transitive, irreflexive and anti-symmetric ordering on the set of actions in the relation. A model of behaviour like this is presented in [178]. Intuitively, this relation represents the facts that no action can cause itself (irreflexive); if an action causes another action then the second action cannot cause the first (anti-symmetry); if one action causes another which causes a third action then the first indirectly causes the third (transitivity).

We note here that it is not the case that schemas defining actions are treated as first class citizens in Z. For example, using the model of behaviour as an ordering relation on action templates, we would like to state that for all action pairs in the behaviour relation the postcondition of the first implies the precondition of the second. This might be represented as:

$$behspec == \{ar : action \leftrightarrow action \mid (\forall a_1; a_2 : action \mid (a_1; a_2) \in ar \bullet post\ a_1 \Rightarrow pre\ a_2)\}$$

Postconditions and preconditions apply to operation schemas in Z. Unfortunately, it is not the case that declaring a reference to a schema, *e.g. x: SchemaX* declares *x* to have all the declarations and predicates associated with the schema *SchemaX*. Rather, declaring *x: SchemaX* states that *x* has the type given by the declarations in *SchemaX* only, *i.e.* the predicates are not included. Thus if *SchemaX* was defined as:

SchemaX

$$c : \mathbb{N}$$
$$d : \mathbb{R}$$
$$c \leq d$$

Then declaring *x: SchemaX* would model *x* as an unordered pair of natural and real numbers with references *c* and *d* respectively. Further, if operations on *x* were performed in some other schema then there would be no preventing the value of *c* becoming greater than *d*, since the predicates are not given. >From this discussion, it is apparent that it is not possible to establish postconditions and preconditions to schema references.

4.2.3 State (of an object)

The state of an object at a given instant in time is the information that influences the set of all sequences of actions in which the object can take part. Knowledge of state does not necessarily allow prediction of the

sequence of actions that will actually occur, since behaviour may include many possible series of actions. Actions can cause modifications to the state of an object.

Formalisation in LOTOS

The state of an object is governed by the behaviour expression defined in the object template from which the object was created and by the current bindings of any existing local variables.

It should be noted that this definition incorporates both the idea of state as represented by the labelled transition system of LOTOS, *i.e.* as placeholders over which actions are defined, as well as state as it might be represented through the values of variables associated with processes, *e.g.* variables contained in the value parameter list of a process definition. Examples of this latter model of state are given in [144].

Formalisation in Z

Classic Z: A binding of the state variables declared in the state schema(s) associated with the object template used for calculating preconditions.

Abstract Z: the state of an object can be represented at many different levels of abstraction depending upon how low a level of information model is considered. The most basic model would consider state information simply as a basic data type. The value that this basic type is bound to would then represent the state of the object. A more structured approach to representing state information would be to represent it as a schema type as with classical Z. The values that the variables in the schema are bound to would then correspond to the state of the object.

4.2.4 Interface

The notion of interface is central to distributed systems, particularly those based on object technology. Interfaces are behaviours, or more precisely, interfaces are the behaviours of objects when focus is placed on a subset of the interactions of those objects. Given that objects are encapsulated, the design of interfaces is fundamental to distributed systems success. That is, objects can only interwork with one another provided their interfaces permit them to do so. We shall investigate interface interworking and the checks that are needed in chapter 7.

Formalisation in LOTOS

An interface is represented in LOTOS as the abstraction of the behaviour of an object. This abstraction focuses on a subset of the observable actions of that object that are to form the observable behaviour of the interface. All actions outside of this set are treated as internal actions in that interface. The behaviour exhibited by an interface is the behaviour exhibited by an object when all actions outside the set of observable actions that compose the interface are made internal. As all observable actions of an object in LOTOS require gates with which to synchronise with the environment, the subset of observable actions is usually achieved by partitioning the gates given in the process definition associated with the object. In order to obtain an interface, hiding the gates not required for the interface under consideration can be achieved. Alternatively, synchronising on only a subset of the gates associated with an object can be used. In this case, actions occurring at those gates in the process definition not in the set synchronised with, may be regarded as actions internal to the object as far as the environment synchronising on those gates making up the interface is concerned.

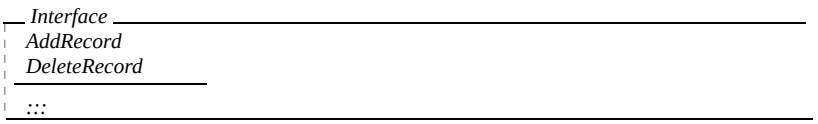
It should be noted that this definition requires that the interfaces of an object use different gate names, *i.e.* it is not possible to distinguish between interfaces that use the same gate.

It should also be pointed out that this modelling of interface is to a certain extent contrived in LOTOS. Interfaces can be modelled but they are not an inherent feature of the language, *i.e.* LOTOS models behaviour and not especially interfaces or the behaviour of interfaces.

Formalisation in Z

Classic Z: An abstraction of the behaviour of an object obtained by identifying the operations associated with that object that are to form the substance of the interface. In all remaining operation schemas all inputs and outputs are hidden, and the occurrences of the operations defined in these operation schemas are regarded as internal actions, *i.e.* they do not require or involve the participation of the environment of the object. The resulting Z text representing that object is an interface template. Any instance of an interface template is an interface.

We note here that it is not possible to simply group operation schemas together to form an interface through schema inclusion. For example, consider the following Z fragment:



The problem with this approach is that it is likely that the two operations would have different and potentially opposite predicates, *e.g.* one adds a record to a database, the other deletes a record from a database. As a result, the predicates would not be well defined. It is of course possible to use predicates to guard against such contradictions from arising, *e.g.* if the record is or isn't in the database then only delete or add can be performed respectively. Such an approach may not always be possible though, *e.g.* when operations are not necessarily contradictory but result in different changes to the state.

Another problem with this approach is that the schema *Interface* does not consist of a collection of individual operations. It is a single schema with declarations and predicates given by all of the included schema declarations and predicates. An alternative approach is to include references to operation schemas. This might be represented as:

The problem with this approach though (see section 4.2.2) is that the semantic information of the schemas *AddRecord* and *DeleteRecord*, *i.e.* their associated predicates, is lost.

Abstract Z: the modelling of interfaces depends predominantly upon how the behaviour of an object is represented. If no distinction is made between observable and internal actions in a behaviour specification, then it is meaningless to talk about interfaces in the specification. Suppose that we can make the distinction between observable and internal actions, and that a behaviour is represented by a relation between actions. An interface then corresponds to that relation with its domain and range modified. This modification requires a subset of the observable actions to be identified. All other observable actions in the relation are then made internal.

4.2.5 Activity

Activities are defined in ODP [108] as single headed acyclic graphs of actions, where each action in an activity is made possible by the occurrence of all immediately preceding actions. From a modelling perspective, activities provide features for reasoning about and specifying behaviours. For example, as we shall see in section 4.4, concurrency models can be interpreted in terms of starting new or joining old sequences of actions in activities.

Formalisation in LOTOS

LOTOS is particularly suited to modelling activities due to the nature of the temporal ordering of events. The actual modelling of an activity depends to a great extent on the interpretation of *action* in *graph of actions*. In section 4.2.1 it was stated that actions were "things that happen". This then means that an

activity is a collection of things that happen in a specified order. Further the things that happen in an activity cannot happen again in that activity. In section 4.2.1 we showed that actions in LOTOS are modelled as the occurrence of event offers.

The consequence of this interpretation of activity is that all LOTOS processes represent activities. This includes recursive processes and processes that, when instantiated, offer several events at the same time. We illustrate these points with an example. Consider the following LOTOS fragment:

```
process P[a,b]:noexit:=
     a;P[a,b] [] b; P[a,b]
endproc (* P *)
```

Once instantiated, this process offers a choice of actions, **a** or **b**. >From this, it might be construed that the process as a whole would violate the single headedness of activities. Given that actions are "things that happen" though, this is not the case since neither **a** nor **b** as event offers actually happen. That is, they only happen when the environment synchronises with them at gates **a** or **b**. As all events in LOTOS are interleaved, the environment cannot synchronise at both gates at the same time. Hence only a single event offer becomes an event, *i.e.* only one thing happens.

The recursive call does not violate the acyclic properties of activities since recursion in the sense of a behavioural loop does not exist in LOTOS. For example, should event **a** occur then it is not the case that a loop is made back to the same node in the graph. Rather, the graph is represented by collections of sequences of **a** or **b** actions. This is understandable if one considers action **a** as **insert_coin** say and process **P** as a money swallowing device. In the real world the insertion of a coin is different each time a coin is inserted into the device. Hence **insert_coin** is the action template for inserting a coin into a machine, but the action instances of inserting coins into machines are always different.

Given a slightly different model of an activity where it is action templates (see section 4.3.11) in the graphs, as opposed to actions, (*i.e.* action occurrences), then modelling an activity requires specification styles to be adopted. For example, the single-headedness of the activity has to be modelled directly. This might be achieved through single event offers that prefix behaviour expressions that do not make possible the single event offer again, *e.g.* through process instantiation. In this case, the single event offer represents the head of the activity and the other behaviour expressions represent the rest of the graph of actions in the activity.

It should also be noted that despite the obvious features of LOTOS for modelling activities, *i.e.* the temporal ordering of events, activities themselves are not first class citizens in LOTOS. That is, the language allows for the modelling of activities but does not provide features for treating them as communicable values or for reasoning about them in any way. This is normally done outside of the specification, *e.g.* when testing.

Formalisation in Z

Classic Z: The notion of an activity as a single headed directed acyclic graph of actions does not exist directly in the Z language. However, the concept of an activity may be modelled to some extent by noting that if action *x* precedes action *y* in some activity then the postcondition of action *x* must imply the precondition for action *y*.

Abstract Z: Activities can be modelled abstractly in Z through relations between actions. To illustrate this we introduce (model) particular ordering relations used in the definition of an activity. The most elementary of these is a digraph. A digraph is a set of actions (*as*) with an ordering relation (*or*) between them. This ordering relation might for example relate postconditions and preconditions of action pairs. This can be represented as:

$$digraph == \{as : \mathbb{P}\,\mathrm{action};\ or : \mathrm{action} \leftrightarrow \mathrm{action} \mid (\mathrm{dom}\ or \cup \mathrm{ran}\ or) \subseteq as\}$$

A directed acyclic graph (*dag*) is a directed graph that contains no cycles. This can be represented as:

$$dag == \{as : \mathbb{P}\,\mathrm{action};\ or : \mathrm{action} \leftrightarrow \mathrm{action} \mid (as, or) \in digraph \wedge \mathrm{disjoint}\ \langle or^{+}; \mathrm{id}\ \mathrm{action}\rangle\}$$

Here or^+ represents the transitive closure of the ordering relation and *id* is the identity relation on a set. The above states that no node can be reached in one or more steps from itself. Thus there are no cycles in the graph.

A connected directed acyclic graph (*condag*) is a directed acyclic graph that does not have separate subgraphs. This can be represented as:

$$condag == \{as : \mathbb{P}\,action;\ or : action \leftrightarrow action \mid (as; or) \in dag \wedge (or \cup or^{\sim})^* = action \times action\}$$

Here $or^{\sim}$ represents the relational inverse of the directed edge relation. Thus $or \cup or^{\sim}$ describes edges where the nodes are joined in both directions and $(or \cup or^{\sim})^*$ is the reflexive transitive closure of the edge relation, *i.e.* it relates all nodes reachable by zero or more steps along the edges. $action \times action$ is the set of all pairs of sets of actions. The condition therefore states that all nodes can be reached from all others in zero or more steps, hence the graph is connected.

Finally an activity corresponds to a connected directed acyclic graph of actions that is single headed.

$$activity == \{as : \mathbb{P}\,action;\ or : action \leftrightarrow action \mid (as; or) \in condag \wedge (\exists a : action \bullet \{a\} = as \setminus ran\ or)\}$$

We shall see in section 4.4 that activities are used to reason about certain types of behaviours.

4.2.6 Environment (of an object)

When modelling, abstractions of some real world system or domain are built. Needless to say, it is impossible to specify the whole world or domain. Yet, frequently we are interested in how our model will interact with the "rest of the world". This is especially the case if we wish to develop open systems, which as argued in section 1.1.1, are one of the main advantages of distributed systems since they facilitate issues such as system evolution and dynamic resource discovery. To enable these issues to be addressed the environment of an object is often considered. This may be regarded as the part of a model which is not part of that object or everything outside of the model. This can either be left completely unspecified or specified to some extent.

Of course, it is possible also to have closed systems. These may be regarded as systems without an environment, or more exactly, systems where the environment does not effect the system behaviour. Given our interest in distributed systems and the advantages offered by openness presented in section 1.1.1 we focus here on open systems.

Formalisation in LOTOS

In LOTOS, the environment of an object is given by the environment of the specification and by the other behaviour expressions that are composed with that object in the specification. The environment of a specification is empty, *i.e.* it is a closed system, if the specification is not parameterised with gates. It should be noted that it is also possible to parameterise LOTOS specifications with data types, specifically those existing in the LOTOS standard library [101]. This can be construed as information provided by the environment that may be used to influence the behaviour exhibited by the specification. This influence is less direct than the influence of the environment through gates though.

Formalisation in Z

Classic Z: The environment of an object in a Z specification is described in terms of the object's input and output. Input to an object comes from the environment. Output of an object goes to the environment. The environment of an object can either be specified directly or left unspecified. If it is unspecified then the occurrence of operation schemas producing outputs or requiring inputs may occur with the environment of the specification as a whole either providing the inputs or receiving the outputs respectively. If the environment of an object is specified, however, then this implies that for each operation schema associated with the object there exists another operation schema (possibly associated with another object) that requires

inputs or outputs of the same type as the object under consideration. These two operation schemas are then conjoined, with the inputs/outputs of the operation under consideration being renamed as the outputs/inputs of the operation representing the environment.

The environment of an object may also be given by variables referenced by an object that have a global scope, *e.g.* those found in axiomatic descriptions.

Abstract Z: If specified, the environment of an object in an abstract Z model corresponds to all of the other objects that are composed with that object. If left unspecified, then the environment of an object is similar to Z in its classic usage, *i.e.* the environment provides inputs to actions and accepts outputs from actions.

4.2.7 Communication

Communication is fundamental to modelling multi-object systems, *i.e.* where the models we make are of objects that send messages to one another. To function correctly, the messages that are sent should obviously have an understood form. Moreover their content, *i.e.* the information they convey, should be understood also. This issue is considered in more detail in chapter 7.

Formalisation in LOTOS

Any synchronisation between processes in LOTOS is a communication. The different synchronisation possibilities in LOTOS, *e.g.* pure synchronisation, value passing and value establishment, offer different features for modelling communication. Value passing allows specific values to be sent between processes. These values can be considered as information items. Value establishment allows ranges of values to be sent between processes. Pure synchronisation does not pass values as such, but the synchronisation event itself can be interpreted as a form of communication.

It is also possible to model asynchronous communication in LOTOS. One way this can be achieved is through having processes communicate through another process representing the connecting medium. Examples of this model are given in section 6.2.5 where issues related to unreliable media and latency are specified.

Formalisation in Z

Classic Z: Communication may be modelled in Z through inputs and outputs to operations. Inputs to and outputs from operation schemas are normally considered as communications with the environment of an object. Since communication occurs between objects, the environment of an object (see 4.2.6) must be specified to model communication. Communication is then achieved by firstly normalising the operation schemas associated with the interacting objects and then conjoining them, with the outputs of one operation being renamed as the inputs to the other operation schema. This modelling of communication requires that the inputs and outputs of the associated operation schemas are of the same type.

Alternatively, the occurrence of operation schemas that reference variables global to the specification represents communication. The value of the global variable following the operation occurrence is communicated to all other operation schemas referencing that variable.

Abstract Z: Communication may also be modelled abstractly in Z. The form of this modelling depends to a great extent on how actions are modelled. Given a model of actions as in section 4.2.1, then communication may be modelled through conjoining the schemas representing action signatures and adding a predicate that the inputs of one schema are equal to the outputs of the other schema (and vice versa).

4.2.8 Location in Space

Given the potential spatial separation of distributed systems and the possible mobility of objects in the system, the location in space at which actions occur is interesting for modelling purposes.

Formalisation in LOTOS

The concept of space is not considered primitive in LOTOS. The location in space at which an action occurs can only be given in LOTOS in terms of the specification model rather than the real world system being modelled. One way that this might be achieved is through the use of an Act One sort representing location in space. Including instances of this sort in the action denotations associated with interacting objects can then be used to limit the interactions that occur between processes.

Formalisation in Z

Classic Z and Abstract Z: As with LOTOS, the concept of space is not considered primitive in Z. The location in space at which an action occurs can only be given in Z in terms of the specification model rather than the real world system being modelled. Thus a location in space might be introduced as a Z type. Through this, relations can be specified associating actions (operation schemas or abstract models of actions) with specific locations in space. This then makes it possible to reason about locations in space at which actions can occur.

4.2.9 Location in Time

The location in time at which an action can occur represents the fact that certain actions may have timing constraints on their occurrence. Such information is needed when composing systems together to ensure that the desired behaviours occur at the desired times.

Formalisation in LOTOS

LOTOS abstracts away from the concept of time, only considering temporal order so there is no *absolute location in relative metric time*. Location in time would be possible however, if an extended form of LOTOS such as E-LOTOS [112] were used with time aspects incorporated.

Formalisation in Z

Classic Z and Abstract Z: As with location in space, the concept of time is not considered primitive in Z. The location in time at which an action occurs can only be given in Z in terms of the specification model rather than the real world system being modelled. Thus a location in time might be introduced as a Z type that can be associated with given actions, *e.g.* through some relation. Through this, quantification over the time at which actions can occur can be achieved. This exact form of this quantification depends to a great extent on the timing policy in effect. For example, two possibilities might be that the relation signifies the actual time at which the action should occur or the latest time at which the action can occur.

4.2.10 Interaction Point

An interaction point represents a location in time and space at which a set of interfaces exist. That is, at a given location in time, an interaction point is associated with a location in space. Several interaction points may exist at the same location in time and space.

Formalisation in LOTOS

A gate with a possibly empty list of associated values. Mobile interaction points may be modelled to a limited extent in LOTOS through modelling locations in time and space as Act One sorts. Synchronisations between processes can be restricted by the value of these sorts. Further, these sorts can be passed around between processes to allow other synchronisation possibilities. It should be noted that this is a contrived form of interaction point modelling in LOTOS. LOTOS was not designed with such dynamicity in mind. This feature is available in other languages such as the Pi-calculus [143] and the current E-LOTOS standard [112].

Formalisation in Z

Classic Z and Abstract Z: The concept of interaction point depends very much upon the definitions of interaction and locations in space and time. Since Z does not make a true distinction between internal and observable actions, nor provide specific features for modelling locations in space and time, the formalisation of interaction point in Z can only ever be done in a contrived manner. One way that this might be achieved is through including specific types representing locations in space and time in the actions (operation schemas or abstract representation of actions) that represent the observable actions of the interface. Through this, reasoning about interaction points can be achieved, *e.g.* through quantification over the space or time variables.

4.2.11 Object

There are almost as many definitions of what an object is as there are books on object technology. An object is typically a model of some thing or entity. ANSA [192] classifies a *thing* as anything that can be *named*. Objects are distinct from one another. Objects are characterised by their behaviour and state. Emphasis may be placed on either behaviour or state. When emphasis is on behaviour, an object is said to perform functions and offer services. Objects can perform more than one function. A function can be performed by a collection of objects. Functions are performed through interacting with an object at its interaction points.

Formalisation in LOTOS

An instantiation of a LOTOS process definition which can be uniquely referenced. Typically this is achieved through having a parameter in event offers that reference the object with which the event is attached. Examples of this style of specification are given in [145, 146].

Formalisation in Z

Classic Z: An object may be described in Z by a collection of specification fragments. These fragments should contain a collection of operation schemas (representing the interface to the object) referring to some state schema(s) (representing the state of the object). The specification fragments should also have some means whereby they can be uniquely referenced (representing the identity of the object). This can be achieved through having an identifier in the state schema(s) of the object that remains constant in all operations defined for that object. Finally, there must exist a valid initial state for that object. This can be achieved through an initialisation schema that gives legal bindings to the variables declared in the state schema with a predicate that ensures the object identifier is unique within the specification.

Abstract Z: it is possible to represent objects at many levels of abstraction. It is quite possible to treat basic data types as objects. This approach enables direct reasoning about object manipulation, *e.g.* passing, creation and deletion of objects. Whilst valid at a high level of abstraction, this approach is limiting since the data types themselves do not have properties that one would normally associate with objects, *e.g.* encapsulated state, identity and interfaces. At a lower level of abstraction an object can be represented by a collection of interfaces with behaviour that relates them. Further, objects should have some means whereby they remember the effect of their behaviour, *i.e.* state.

4.3 Requirements on Specification Languages

The above concepts reflect directly features that one would expect to find when modelling systems containing objects. To model systems of objects it is necessary to introduce further concepts that describe objects and their associated characteristics. For example, object template, instantiation, class and behaviour compatibility are all concepts used when describing objects. These further concepts may be regarded as desirable features that may or may not exist in our specification languages should we wish to model systems of objects.

4.3.1 Composition of Objects

The notion of composition is crucial to distributed systems. Modelling isolated objects is of little interest. Modelling distributed systems as collections of objects is only viable if these objects can interact. Composition makes possible such interactions. We shall investigate different forms of composition in chapter 7.

As well as this, composition is also a powerful design tool. The ability to take two or more objects and combine them to form a new object at a different level of abstraction is at the heart of bottom-up development strategies.

Formalisation in LOTOS

A composite object is an object described through the application of one or more LOTOS combination operators. These include:

- interleaving operator (|||)
- parallel composition operators (|| and |[gate-list]|)
- enabling operator (>>)
- disabling operator (**[>**)
- choice operator (**[]**)

Formalisation in Z

Classic Z: Composition of objects is not a feature explicitly offered by the Z language, due amongst other things to the lack of encapsulation. However, it is possible to model some characteristics of composition through schema inclusion and redefinition of operations through promotion.

Abstract Z: Having an abstract model of objects as given in section 4.2.11 allows for a multitude of composition possibilities. For example, composition may be regarded as a relation between objects where the exact form of the composition is determined by the predicates associated with the relation. We shall see in the following chapter examples of how an abstract representation of objects and the interfaces that are associated with them allows composition based on syntactic, behavioural and non-functional aspects.

4.3.2 Composition of Behaviours

As with composition of objects, composition of behaviours is a powerful tool when designing distributed systems. The ability to take two or more behaviours and combine them to yield a new behaviour is fundamental to (distributed) system design.

Formalisation in LOTOS

The composition of the behaviour expressions associated with the component objects in the creation of a composite object through composition. The operators for the composition of behaviours are the same as those for the composition of objects with the inclusion of the action prefix (sequencing) operator (**;**).

Formalisation in Z

Classic Z: As a behaviour in its most degenerate case may be considered as a singleton action, and an action in Z is the performance of an operation defined by an operation schema, composition of actions equates to the combination of operation schemas in Z. Operation schemas may be combined in several ways in Z, such as:

- schema calculus

- schema composition (**;**)
- overriding ($\oplus$)
- piping ($\gg$)

It should be pointed out that the last of these operators does not have a well defined semantics in Z and is unlikely to be in the Z standard [106] which is currently being developed. The same effect can be obtained through conjunction and renaming of schema variables though.

Abstract Z: at a more abstract level, composition of behaviours can be achieved in many ways in Z. Given that a behaviour might be represented abstractly by a relation, composition of behaviours can be modelled in as many ways in Z as there are ways for composing relations.

4.3.3 Decomposition of an object

If composition of objects represents a bottom-up development strategy then decomposition of objects represents a top-down development strategy. It should be pointed out that the effects of composition and decomposition are one and the same when from the point of view of the specification. That is, composition takes two or more objects and combines them to form a new object. Decomposition can loosely be regarded as the reversal of this process, *i.e.* a composite object is decomposed into its constituent parts. Whichever of these approaches are taken, the result is the specification itself. Thus it is not possible to take an arbitrary specification and determine whether it was developed through composition or decomposition approaches.

Formalisation in LOTOS

The expression of a given object as a *composite object*. There does not exist features in LOTOS to decompose an object into a form that is not the same as the existing form, *i.e.* how the object was composed in the first place. Such issues are normally dealt with when refinement (see section 4.3.6) of a specification is done.

Formalisation in Z

As with LOTOS, decomposition in Z is given by the expression of a given object as a *composite object*. There does not exist features in Z to decompose an object into a form that is not the same as the existing form, *i.e.* how the object was composed in the first place. Such issues are normally dealt with when refinement (see section 4.3.6) of a specification is done.

4.3.4 Decomposition of a behaviour

Decomposition of behaviours allows a structured approach to design. Having identified a specific functionality that should be fulfilled by a component, decomposition of behaviour allows this functionality to be broken down into parts.

Formalisation in LOTOS

Decomposition of behaviour in LOTOS is given by the decomposition of the behaviour expressions associated with the component objects in the creation of a composite object through composition (see 4.3.1). The operators uncovered during decomposition of behaviour are the same as those for the composition of behaviour. This due to the discussion in section 4.3.3.

Formalisation in Z

As with LOTOS, decomposition of behaviours in Z results in the uncovering of the same composition operators as was used in the process of composition of the behaviour due to the discussion in section 4.3.3.

4.3.5 Behaviour Compatibility

Behaviour compatibility is a relation between behaviours. An object is behaviourally compatible with a second object when the first object can replace the second object without the environment being able to notice the difference in the object's behaviour on the basis of defined criteria.

Behavioural compatibility may be natural or coerced. In coerced behaviour compatibility the behaviour of the replacing object is modified in some way so that the environment of the replaced object cannot tell the difference between them. In natural behaviour compatibility, no such modification is necessary.

Formalisation in LOTOS

In LOTOS, specific theories have been developed to check for behaviour compatibility. There are no specific LOTOS language syntactic features to construct and ensure behaviour compatibility generally. The LOTOS standard, however, develops the notion of conformance which provides a basis for consideration of behaviour compatibility.

In order to determine whether or not two object behaviours are compatible, the notion of conformance needs to be introduced. Conformance is concerned with assessing the functionality of an "implementation" against its "specification", where here the term implementation may be taken to be a less abstract description of a specification.

If **P** and **Q** are two LOTOS processes, then the statement **Q** conforms to **P** (written as **Q conf P**) signifies that **Q** is a valid implementation of **P**. This means that if **P** can perform some trace σ and then behave like some process $\mathbf{P}'$, and if **Q** can also perform trace σ and then behave like $\mathbf{Q}'$ then the following conditions on $\mathbf{P}'$ and $\mathbf{Q}'$ must be met: whenever $\mathbf{Q}'$ can refuse to perform every event from a given set A of observable actions, then $\mathbf{P}'$ must also be able to refuse to perform every event of A.

Thus **Q conf P** if and only if **Q**, when placed in any environment whose traces are limited to those of **P**, **Q** cannot deadlock when **P** cannot deadlock. Another way of defining this is that **Q** has the deadlocks of **P** in an environment whose traces are limited to those of **P**.

An object can be made behaviourally compatible with a second object after some modification to its behaviour, which might include extending the object's behaviour (adding additional behaviour) or reducing the object's behaviour (restricting the object's behaviour). This process of modifying an object is known as refinement (see section 4.3.6).

Formalisation in Z

Classic Z: Behaviour compatibility is based upon the notion of substitutability in a given environment. Extension is one possible way of achieving this. An extension of a base template may have extra components in the associated state schema, a stronger state invariant, stronger initial conditions and more operation schemas. The operation schemas associated with the extension of the template type may have weaker preconditions and stronger postconditions than the corresponding operation schemas in the base template type.

Abstract Z: Modelling a behaviour as a relation between actions allows behavioural compatibility issues to be dealt with directly. One behaviour (B_1) is compatible with a second behaviour (B_2) when the following condition holds:

$$\mathrm{dom}\, B_2 \subseteq \mathrm{dom}\, B_1 \wedge (\mathrm{dom}\, B_2) \lhd B_1 \subseteq B_2$$

Here we state that the domain of the first behaviour (B_1) is a superset of the second behaviour (B_2), *i.e.* B_1 accepts all B_2 actions plus possibly more. Further, we state that the actions that B_1 has in common with B_2 produce less results that those of B_2. More precisely, we state that restricting the behaviour of B_1 to relate only those actions found in the domain of B_2 does not produce results (relate actions) that B_2 itself could not produce. Intuitively these conditions imply that behaviour compatibility allows more inputs and produces less outputs. Hence in an environment expecting the first behaviour, the second could be substituted without any distinctions being made. We shall see in chapter 7 the limitations of this interpretation when a notion of causality is introduced into the system.

4.3.6 Refinement

Refinement is the process by which a specification is transformed into a more detailed specification. Precisely what is meant by a more detailed specification depends upon the specification language in use. Typically, some form of behaviour compatibility between a specification and its refinement should exist.

Formalisation in LOTOS

Refinement is the process by which an object may be modified, either by extending or reducing its behaviour or by a combination of both, so that it conforms to another object. Letting **P** and **Q** be LOTOS processes, an extension of **P** by **Q** (written as **Q extends P**) means that **Q** has no fewer traces than **P**, but in an environment whose traces are limited to those of **P**, then **Q** has the same deadlocks. A reduction of **P** by **Q** (written as **Q reduces P**) means that **Q** has no more traces than **P**, but in an environment whose traces are limited to those of **Q**, then **P** has the same deadlocks.

Formalisation in Z

Classic Z: Since Z deals with abstractions of systems where data and operations on that data are used to represent the given system under consideration, two main forms of refinement have been identified: *operation refinement* and *data refinement*. In order to refine a specification, the refinement must ensure behaviour compatibility between the specification and the refinement. To account for this, certain conditions exist to ensure that a Z specification refinement produces a valid more detailed specification. These are the *safety* and *liveness* conditions. The safety condition on the refinement of a specification is that any circumstance acceptable to the specification must be acceptable to the refinement. The liveness condition on the refinement of a specification is that for any circumstance acceptable to the specification, the behaviour of the refinement must be identical to the specification.

The safety and liveness conditions need to apply to both the operation and data refinements.

Abstract Z: Refinement of behaviour modelled as a relation can be done in many ways. Examples of these include modifying the domain or range by adding or removing actions and modifying the relation itself by including new relationships between actions. These modifications to the behaviour must ensure that behavioural compatibility (see section 4.3.5) exists between the refined and original behaviours.

4.3.7 Type of an <X>

A type is a predicate that characterises a collection of <X>s where an X may be an object, interface or action. Types are typically used to restrict interactions between objects.

Formalisation in LOTOS

Types that can be written down explicitly in LOTOS for objects and interfaces are template types. There is no explicit construct in LOTOS that will permit the modelling of action types as such. A LOTOS specification consists of a behaviour expression which is itself composed of action denotations (action templates). These action templates either occur as part of the behaviour of the system, in which case their occurrence may loosely be regarded as the action template instantiation, or they do not occur, in which case the action template remains uninstantiated. The action templates themselves may be given by the internal event symbol, **i**, or event offers at gates which may or may not have finite sequence of value and/or variable declarations.

LOTOS does not offer facilities to characterise actions directly, however a limited form of action characterisation is built into the synchronisation feature of LOTOS. That is, it might be considered that synchronised action denotations (action templates) must satisfy the same action type in order for the action to occur. However, LOTOS does not classify the characterising features of these arbitrary action denotations and thus it is not possible to have a formal type for any given action. It might be the case that informally the event offers involved in an interaction are given a cause and effect role, but this is generally not the case. See section 4.2.7.

The internal event symbol may be used to represent an action type, where the common characteristics of this collection of actions are that they have no characteristics.

It should be noted that by stating that the only predicate possible in LOTOS for objects (and interfaces) is that they satisfy their template type, the concepts of type and template type reduce to the same modelling technique in LOTOS. Thus there is no distinction in LOTOS between a type in its broad characterisation sense, and a template type in its more restrictive sense of template instantiation.

Formalisation in Z

An object, interface or action can have many different types. These types depend entirely on the predicate that is applied. As a result it is not possible to formalise a generic notion of type. It can be stated that Z types correspond to sets though and that the characterising predicate of such Z types is given by set membership.

4.3.8 Class of <X>s

The concept of class in computing science has traditionally been synonymous with implementation of objects. Another view of class is as simply a set where the elements of the set satisfy some type. These choices reflect the different interpretations[3] of class: the first being an intensional model, the second being an extensional model. ODP defines class extensionally and we use this interpretation for our formalisation.

Formalisation in LOTOS

The notion of class is dependent upon the characterising type predicate which the members of the class satisfy. Objects, interfaces and actions can satisfy many arbitrary characterising type predicates. A type that can be written down is a template type (see section 4.3.18. When this is the case, the set (class) of objects, interfaces and actions that are associated with that template type is the template class (see section 4.3.19.

It should be noted that by stating that the only classification possible in LOTOS for objects, interfaces and actions is that they satisfy their template type, the concepts of class and template class reduce to the same modelling technique in LOTOS. Thus there is no distinction in LOTOS between a class in its general classification sense, and a template class in its more restrictive sense as the set of instances of a given template type.

Formalisation in Z

Since a type as presented in section 4.3.7 may not be modelled in Z due to its genericity, the modelling of class is also limited. It can be stated though that the members of a class should be members of the set that defines the type.

4.3.9 Subtype/Supertype

One type is a subtype of another when it satisfies the same predicate as the first. Given our interest in specifying systems that may evolve, subtyping provides a mechanism for reasoning about acceptable substitutes. Further, since types are frequently used to restrict the interactions that can occur between systems (objects), subtypes should possess all of the interactions and more. In this case, the satisfies relation corresponds to “responds to these messages”.

Formalisation in LOTOS

As the types that can be written down in LOTOS for objects, interfaces and to a lesser extent actions are template types (see section 4.3.18), a subtype relation in LOTOS is a relation that may exist between template types. In LOTOS, however, there exists no feature to write down subtyping relations directly. If subtyping is required then extension can be used to give a subtype relation based on substitutability. However, this is not a feature explicitly provided for in LOTOS.

[3] A prime example of the confusion in terminology that exists in object technology today.

Formalisation in Z

Subtypes and supertypes correspond to subsets and supersets respectively in Z. The exact form of this set membership depends entirely on how the set defining the type is modelled. As stated, it is often the case that subtyping is based on syntactic aspects of the type. In such cases, subtyping between two sets requires that the syntactic aspects of the elements found in one set are compatible with the syntactic aspects of the elements found in the other set. Here compatibility for interface signatures say might be based on similar names for actions and compatible input and output parameters. We investigate this issue in more detail in the following chapters.

4.3.10 Subclass/Superclass

One class is a subclass of a second class and the second a superclass of the first when the type associated with the first is a subtype of the type associated with the second. Subclasses are by definition subsets of their superclasses given this extensional interpretation of class.

Formalisation in LOTOS

As the types that can be written down in LOTOS for objects, interfaces and to a lesser extent actions are template types (see section 4.3.18), a subclass relation exists between two classes when a subtyping relation exists between their corresponding template types.

Formalisation in Z

Subclasses and superclasses correspond to the subset and superset relationships respectively in Z. The exact form of this set membership relation depends entirely on how the set defining the type is modelled.

4.3.11 <X> Template

To model systems of objects, interfaces and actions, a description of these concepts must be made. These descriptions may be captured in a template. Templates capture common features of collections of objects, interfaces and actions in sufficient detail that when instantiated, the object, interface or action respectively will be generated.

Formalisation in LOTOS

- **Object Template**: A process definition with some means by which it can be uniquely referenced once instantiated. This can be achieved for example, by interacting with some other process to obtain a unique identifier (sort), or by being assigned a unique identifier upon instantiation.

  With regard to combination of object templates in LOTOS there are no existing combination operators except for a limited form of scoping using the LOTOS “where” clause.

- **Interface Template**: Any behaviour obtained from a process definition by considering only the interactions at a subset of the gates associated with the process definition. This subsetting of the gates is achieved by hiding the gates not required for the interactions under consideration.

  With regard to combination of interface templates in LOTOS there are no existing combination operators except for a limited form of scoping using the LOTOS “where” clause.

- **Action Template**: An action denotation where an action denotation may be either an internal-event symbol, a gate-identifier or a gate-identifier followed by a finite sequence of value and/or variable declarations.

The definition here of action denotation is contrived as LOTOS does not really support the concept of an action template. In LOTOS, possible behaviours are specified by giving action denotations combined in some form. To relate a template to an action denotation is the closest that can be achieved in LOTOS.

However, the text of Part 2 requires an action template to group the characteristics of actions. This is not part of LOTOS as event offers (action denotations) exist in isolation and it is not possible to collect them and apply a template to characterise them.

Composition of action templates may loosely be likened to synchronisation with value passing or value generation. In this case, two (or more) action templates agree on a common action template for the synchronisation to occur, *i.e.* an action template with the common characteristics of all of the action templates involved in the synchronisation (composition).

Formalisation in Z

Classic Z:

- **Object Template:** Fragments of a specification that represent a state, have a unique (immutable) identity that can be referenced, and have an associated set of operation schemas that act on that state. If the object template is a generic one, the precise form of template will only be given when the types of the parameters are given.
- **Interface Template:** A set of operation schemas derived from the Z text representing an object template in the way described under the interpretation of interface (4.2.4). If the object template is a generic one, the precise form of interface template will only be given when the types of the parameters are given. Interface templates may be combined using the Z operations for schema combination.
- **Action Template:** An operation schema. Action templates may be combined using the Z operations for schema combination. If the action template is a generic one, the precise form of action template will only be given when the type of the parameterising parameters are given.

Abstract Z:

- **Object Template:** An object template may be represented by a collection of interface templates with a behaviour specification that relates the behaviours associated with these interface templates. Since objects have state also (see section 4.2.11) there should exist a model of state within the object template, *e.g.* as a variable. This can be used to remember the effect of the actions that have occurred. There should also exist some means whereby the objects instantiated from the template can be uniquely referenced.
- **Interface Template:** An interface template may be represented by a collection of schemas that capture the syntactic aspects of the functionality offered by the interface, *i.e.* the signature. A behaviour specification must also be provided that shows the effect of invoking the syntactic aspects of these interfaces. In addition, there should be some means whereby the effect of the actions occurring can be recorded, *i.e.* state. We illustrate this with a Z fragment that uses the representation of behaviour given in section 4.2.2. Here *Signatures* are schemas used to capture the syntactic structure of actions in the interface and *history* is a model of the states of the interface. It should be noted that it might be the case that we have only one sequence of states associated with a given object or interface. If this is the case then the history can be represented directly as:

$$history == \mathrel{_{seq}} State$$

It is also possible to model more complex states that might be associated with an object or interface, *e.g.* if we have more than one state at any time. This might be the case if the object or interface is itself distributed. In such a case the state of the whole object or interface may be composed of several partial states. Further it may not always be possible to establish the state of the object or interface at one time, *e.g.* the partial states may be perpetually changing and checking their values simultaneously is impossible due to their distribution. Such complexity can be reflected in the Z text modelling history, *e.g.* as sequences of sets of states.

>From this, a model of an interface template may be represented as:

Here *predicates* represents the predicates necessary to relate the signatures and states to the actions in the behaviour specification.

- **Action Template:** Instantiation of an abstract model of an action template can take several forms. One possible form is simply a modification of the states associated with the object the action template is part of. For example, using the above model of states as a sequence of sets of possible state sequences, then the instantiation of an action template would require that the state required for the action to occur is in the current set of sequence of possible states (*history*). Further, the range of the action relation, *i.e.* the state following the action occurrence, is in the next set of state sequences of the history.

4.3.12 Interface Signature

An interface signature represents the syntactic aspects of an interface. This includes the operation names along with the input parameters and expected return parameters of the operations. Often an indication of direction of the parameters is given explicitly, *e.g.* through labelling them as **in, out** or **inout**.

Formalisation in LOTOS

An interface signature as a set of action templates associated with the interactions of an interface is represented in LOTOS by a set of action denotations. The members of this set are those action denotations that require synchronisation with the environment in order to occur.

It should be noted that LOTOS does not provide explicit features for describing interface signatures as such. That is, event offers generally do not simply capture the syntactic aspects of the action. They define the actions themselves. It is possible to adopt a convention whereby event offers are treated as syntactic devices for other behaviours though. We illustrate this with the following LOTOS fragment:

```
process P[g,h](s: State):noexit:=
     g !modify ?x: Nat;
     P[g,h](modify(x,s))
endproc (* P *)
```

Here the operation name is given by *modify* and the input parameters are of type natural number. The semantics (behaviour) attached to this event offer is then given by the modifications to the state given in the process instantiation.

It should be pointed out that LOTOS does not allow for any means to check directly that interface signatures are compatible with one another. We provide an Act One approach that allows interface signature checking in chapter 7.

Formalisation in Z

Classic Z: The normalised declarations found in those schemas that make up part of an interface.

Abstract Z: A collection of schemas that capture the syntactic aspects of the observable actions associated with the interface. We shall see in the following chapter that ODP prescribes specific forms for interface signatures in the computational viewpoint.

4.3.13 Instantiation of an <X> Template

Having modelled objects and interfaces in a template, instantiating that template is necessary to produce the respective object or interface. Instantiation may involve actualisation of parameters.

Formalisation in LOTOS

- **of an Object Template**: The result of a process which uses an object template to create a new object in its initial state. This process involves the actualisation of the formal gate list and formal parameters of a process definition by a one-one relabelling from a specified gate list and list of actual parameters. The features of the object created will be governed by the object template and any parameters used to instantiate it.
- **of an Interface Template**: The result of a process by which an interface is created from an interface template. The interface created can thereafter be used by the object it is associated with to interact with the environment. The features of the interface created will be determined by the interface template and any parameters used to instantiate it.
- **of an Action Template**: This is given as action occurrence in LOTOS. This may involve the rewriting of Act One expressions.

Formalisation in Z

Classic Z:

- **of an Object Template:** Initialisation of the Z text representing the object template. This is often provided for explicitly in Z by an initialisation schema. The performance of this initialisation schema must satisfy the condition that there exists a valid state for the object after this initialisation schema occurrence, *i.e.* a state which satisfies any invariants that may be present. These invariants may refer to other objects.
- **of an Interface Template:** Initialisation of the Z text corresponding to an interface template. An interface template may be obtained as shown in 4.2.4. It thus follows from this that the instantiation of an interface template and an object template are both achieved the same way in Z, via initialisation of the associated Z text.
- **of an Action Template:** The performance of an operation specified in an operation schema.

Abstract Z:

- **of an Object/Interface Template:** Instantiation of an abstract model of an object or interface template may be modelled through an operation schema. This schema should provide a state that is valid for that object or interface template. A valid state here depends on how the states are modelled. Given the model of states in section 4.3.11, *i.e.* as a sequence of sets of possible state sequences, then a valid state would be one of those in the head of the sequence. In the case of object templates, the operation schema should also provide an identifier through which the resulting object can be uniquely referenced.
- **of an Action Template:** Instantiation of an abstract model of an action template can take several forms. Given a model of actions as presented in section 4.2.1 and objects as presented in section 4.3.11, the instantiation (occurrence) of an action template requires that the current state of the object, *i.e.* the current position in the history sequence, contains the state necessary for the action to occur. Further, following the action occurrence the final state must be one of those in the next set of states in the history sequence.

4.3.14 Creation of an <X>

Objects may be instantiated in two ways depending on how they and the system they exist in are modelled. Objects may instantiate other objects by performing actions. This is termed creation. Alternatively objects can be instantiated by some other mechanism. This is termed introduction.

Interfaces may either be created or introduced. Interface introduction is only possible through object introduction.

Formalisation in LOTOS

- **of an object** The instantiation of an object template as part of the behaviour of an existing object.
- **of an interface** As objects and interfaces are modelled the same way in LOTOS (via process definitions), creation of objects corresponds to creation of interfaces. Thus the definition for interface creation is given by the creation of objects as above.

Formalisation in Z

Classic Z:

- **of an Object**: The creation of an object in Z is given by providing a valid initial state for the Z text associated with the object template. That is, by providing a binding of the variables given in the state schema of the object to the initial values that they hold. Often this is provided for explicitly in Z through an initialisation schema. In this case, the action of creation is represented by performing the operation given in the initialisation schema.
- **of an Interface**: The creation of an interface in Z is inherently linked with the creation of objects. That is, when the text associated with an object template is initialised, any interface templates that might be present are initialised also.

In creation, it is necessary to ensure that the identity of the object being created is unique within the specification. This can be achieved through a framing schema with an appropriate predicate ensuring the identities of all of created objects are unique. This framing schema can then be used to promote the initialisation of an object to effect the specification as a whole.

The text given here implies that, because an initialisation schema is given as part of a specification, an object is created. However, in Z there is no notion of the specification actually applying this initialisation schema. This is in fact the introduction of an object, *i.e.* an object is instantiated by a mechanism not covered by the model. It would appear that the notion of initialisation of a Z specification is partly creation (in that an initialisation schema is given) and partly introduction (in that the application of the initialisation schema is not covered by the model).

It is normally the case that a proof obligation follows the application of an initialisation schema to ensure that the object is in a valid initial state.

Abstract Z:

- **of an Object/Interface**: Creation of an object/interface from an object or interface template respectively requires that an action exists in the behaviour specification of an existing — in the sense of already instantiated — object. This action should result in the instantiation of an object or interface template. As discussed in section 4.3.13, instantiation in an abstract Z model is done with an operation schema. It is not possible to represent such operation schemas directly in the behaviour of a given object, *i.e.* a schema defines a type and relations themselves are typed. Hence behaviour as a relation between actions (modelled themselves as relations between states) cannot accept schemas modelling instantiations of templates. To overcome this, functions can be used that relate arbitrary structures (schemas representing templates) to actions, *e.g.* as with observable and internal action signatures in section 4.2.1. Creation then requires that an action modelled this way exists in the behaviour specification of an existing object. The occurrence of this action then results in the instantiation

(creation) of the associated object or interface. In the case of object creation, this action should also establish a new identifier that can be used to uniquely reference the object.

As can be seen, creation can be modelled when using Z abstractly, however this is a particularly contrived representation.

4.3.15 Introduction of an <X>

Introduction is the process of instantiation when it is not covered by the action of objects in the model.

Formalisation in LOTOS

The instantiation of the behaviour associated with a LOTOS specification.

Formalisation in Z

Classic Z: See Creation (of an object) (4.3.14).

Abstract Z: An abstract representation of the introduction of an object may be achieved by an action modelling instantiation of an object template that is not associated with an existing object. It should be noted that introduction does not require contrived features for overcoming the Z type system as was the case with creation (see section 4.3.14), *i.e.* functions that relate schemas to the models of actions in the behaviour specification.

4.3.16 Deletion of an <X>

Given that objects and interfaces can be brought into existence in a model, *i.e.* instantiated, it is also necessary to provide features for their removal from the model. Deletion of an object or interface corresponds to destroying an instantiated interface or object. Interfaces may only be destroyed by the object with which they are associated.

Formalisation in LOTOS

- **of an object** The termination of a process instantiation. This may be achieved through the use of the LOTOS disabling operator, the LOTOS inaction (**stop**) behaviour expression which does not allow for the passing of control, or the successful termination (**exit**) behaviour expression (where passing of control is possible via the enabling operator).
- **of an interface** The process by which the future behaviour of an object is limited to that behaviour which was not deleted for the interface.

Formalisation in Z

Classic Z: It may be possible to have an abstract representation of deletion where a framing schema is used. Deletion can then be modelled as the promotion of an operation to remove an object state and identity from the system as a whole.

It might also be the case that a form of deletion based upon inactivity may be modelled. For example, an object whose associated future behaviours may no longer occur due to invariants being violated may in some sense be considered as deleted. This form of deletion may not accurately capture the definition deletion accurately though, *i.e.* there is no destruction as such.

Abstract Z: Modelling the deletion of an object or interface depends on how the objects and interfaces themselves are represented. If objects and interfaces are modelled as shown in sections 4.2.11 and 4.2.4 [4] then deletion may be modelled by an action whose resultant state is not found in the domain of any other

[4] And hence actions, behaviours and signatures are as shown.

action associated with that behaviour specification. The occurrence of this action is then equivalent to the LOTOS **stop** expression.

4.3.17 Instance of a Type

An <X> that satisfies the type.

Formalisation in LOTOS

- **of an Object Template** An instance of a given object template is represented in LOTOS by an instantiation of that object template or an acceptable substitution for an instantiation of that object template. Here the acceptable substitute should capture the characteristics that identify this type. Thus an acceptable substitute might be another template that is behaviourally compatible with the first. This might be achieved through extension as defined in section 4.3.5. Using this relation guarantees that all characteristics of the type under consideration are included. It might be the case, however, that a weaker form of type satisfaction relation can be found which does not require all characteristics associated with a given template to be included, but some subset of the total characteristics.

- **of an Interface Template** As an interface template is represented the same way as an object template, (*i.e.* a process definition in LOTOS), the above text applies equally well to an interface template, *i.e.* replace all occurrences of object with interface.

- **of an Action Template** An instance of an action template (action denotation) is represented in LOTOS by an action denotation offering an equivalent event.

Formalisation in Z

Classic Z: An instance of a type in Z is represented by an element of the set whose members satisfy the predicate, *i.e.* the type. Given a more specific notion of type, such as template type (see section 4.3.18), an instance of an object or interface type corresponds to the initialisation of the Z text (or an extension of it) representing the object/interface under consideration, such that there exists a valid initial state for that object or interface. Here the characterising predicate is given by the specification text and associated predicates on possible legal bindings (invariants), which must be satisfied by the initialisation schema in order to be classified as an instance of the object or interface type.

As an operation schema gives the characteristics of an action type, an instance of an action type is given by the occurrence of this operation schema or by the occurrence of another operation schema which is an extension of this operation schema, *i.e.* the extension includes the characterising features of the action type.

Abstract Z: The abstract representation of an instance of a type depends on the way the set is defined on which the type is based. It can be stated though that instances of types should satisfy the characterising predicate that defines the type.

4.3.18 Template Type of an <X>

A template type is a predicate defined in a template that holds for all instantiations of the template and that expresses the requirements the instantiations of the template are expected to fulfill.

Formalisation in LOTOS

The notion of a template type as a predicate expressing that an $< X >$ is an instance of a given template, where an $< X >$ may be an object, an interface or an action is not a feature offered by the LOTOS language. This feature is only available to a certain extent outside of LOTOS, *e.g.* when checking the static semantics of a specification. There issues such as valid process instantiations are dealt with, *e.g.* correct number of gates are given and actual parameters match formal ones.

Formalisation in Z

Classic Z: In Z, an object/interface template type is a predicate that an initialisation schema leaves the object and associated interfaces in a valid initial state. Thus all state variables should be bound and all necessary predicates (invariants) satisfied. Often checking an object or interface template type requires a proof to be made.

An action template type corresponds to a predicate that an action template, as given by an operation schema, can occur, *i.e.* is an instantiation of a given operation schema. Thus all instantiations are expected to satisfy the predicates on legal bindings of variables, as given in the operation schema's predicates.

Abstract Z: An abstract model of a template type depends on how the template on which the type is based is represented. If object and interface templates are modelled as presented in section 4.3.11, then instantiations of these templates should provide a valid initial state. The resultant object or interface will have the behaviours, states and signatures as determined by the template and how it was instantiated.

It is possible in an abstract Z model to reason directly about what constitutes a valid instantiation, *i.e.* through predicates in a schema modelling instantiation.

4.3.19 Template Class of an <X>

A template class is the set of all <X>s satisfying a particular <X> template type, *i.e.* the set of <X>s which are instances of the <X> template. This concept is a specialisation of the idea of class. A class is used as a general classification mechanism. A template class on the other hand is more restrictive in that the members of the template class are limited to those instantiated from a particular template, or any of its subtypes, *i.e.* the members satisfy a particular template type.

Formalisation in LOTOS

The template class of an $< X >$ is the set of all $< X >$s that are instances of that $< X >$ template, where an $< X >$ may be an object, an interface or an action.

The notion of the template class of an action is limited in its application to LOTOS, as LOTOS does not provide explicitly for action templates, action template instantiations or action template types.

Formalisation in Z

A template class of an $< X >$ is the set of all $< X >$s that are instances of that $< X >$ template, where an $< X >$ may be an object, interface or action.

4.3.20 Derived Class/Base Class

If a template is an incremental modification of a second template then the template class of instances of the first template is a derived class of the template class of instances of the second template. This incremental modification may take many forms. It might for example involve adding to or altering the properties of the base class.

Formalisation in LOTOS

LOTOS templates can be incrementally modified by extending, enriching and modifying the data types or by modifying the behaviour. Problems arise with the behaviour modifications however, specifically:

- subtyping: non-determinism may be introduced into the system when the initials of the base template and its modification are the same, thus subtyping cannot be guaranteed;
- the need for a redirection of self-reference: any reference to a derived template from a parent template should be redirected to the derived template, which is not always possible.

We illustrate these problems through a simple example. Consider the following LOTOS fragments:

```
process S1[ g ]:noexit:=
       g; S1
endproc (* S1 *)
```

```
process S2[ g, h ]:noexit:=
    g; S2[ g,h ] [] h;stop
endproc (* S2 *)
```

Here process *S2* is very similar to process *S1*. It differs only in the offering of event *h*. It is not the case that we may substitute *S1* for the identical behaviour in *S2* though. For example if we specify:

```
process S3[ g ]:noexit:=
       S1[ g ] [] h; stop
endproc (* S3 *)
```

This is no longer the same as the process *S2* since once an event *g* occurs, event *h* can never occur. The problem stems from the need to redirect self-reference form *S1* to *S3*. This issue is discussed in more detail in [164]. There is no satisfactory solution to these problems in standard LOTOS.

Despite this, Clark and Moreira [43, 147] have shown how it is possible to specify and reason about inheritance and subtyping in LOTOS. Their approach requires that processes representing base classes have **exit** functionality. With this model they have shown how classes can be derived from base classes using various approaches, *e.g.* extensions to or redefinitions of existing behaviours.

Formalisation in Z

Given two templates A and B in Z where A is an incremental modification of B and the instances of A and B are in a derived class/base class relationship respectively, the incremental modifications to B to produce A may include: adding or deleting state parameters; adding, deleting or modifying operations; or strengthening or weakening invariants.

4.3.21 Invariant

An invariant is a predicate that may appear in a specification. In particular, invariants should always be satisfied both before actions occur and after they have occurred.

Formalisation in LOTOS

In LOTOS, the only invariants which can be written down are process definitions. There is no way to attach an invariant to a process definition which is not the process definition itself.

Formalisation in Z

A predicate that a specification always requires to be true. Z allows invariants to be written down directly in schemas and axiomatic descriptions. Often invariants impose restrictions on the possible bindings that the variables in schemas or axiomatic descriptions can take. As such, an invariant is often used to restrict the possible behaviours given in a specification.

4.3.22 Precondition

A precondition is a predicate that a specification requires to be true for an action to occur.

Formalisation in LOTOS

A precondition may be expressed directly in LOTOS using one or more of: sequencing of actions; guards and selection predicates.

Formalisation in Z

Classic Z: The condition on the state of the system before the occurrence of an operation defined by an operation schema and on its inputs such that there exists a possible state after the performance of the operation and outputs which satisfy the postconditions. Z allows preconditions to be written down directly.

Abstract Z: Modelling actions as relations between states allows preconditions to be checked directly. For example, given the model of behaviour in section 4.2.2, then the precondition for any action occurrence is that the current state of the object, with which the action is associated, satisfies the state given by the domain of that action.

4.3.23 Postcondition

Postconditions are predicates that specifications require to be true immediately after the occurrence of actions.

Formalisation in LOTOS

In LOTOS, the occurrence of an action is independent of the state of the system after the occurrence of the action. As such, LOTOS does not provide the means to directly express postconditions.

Formalisation in Z

Classic Z: A predicate which describes the set of states that a given system can be in after the performance of an operation defined by an operation schema. Z allows postconditions to be written down directly.

Abstract Z: Modelling actions as relations between states allows postconditions to be checked directly. For example, given the model of behaviour in section 4.2.2, then the postcondition of any action is implied by the state it relates to.

4.4 Specific Behavioural Concepts

As well as the more elementary notions of behaviour, the ODP-RM introduces more prescriptive behaviours that objects should possess. Specifically, behaviours are prescribed that enable aspects of concurrency and independent behaviours of objects to be specified and reasoned about. These behaviours include the creation of new behaviour as well as the joining of existing behaviours.

4.4.1 Chain of Actions

Computational objects may be associated with specific forms of activities. Of particular importance are those activities connected with chains, where a chain may be regarded as a sequence of actions within an activity where for each adjacent pair of actions, occurrence of the first is necessary for the occurrence of the second action.

Formalisation in LOTOS

Given a model of activity as discussed in section 4.2.5, then a chain may be represented in many ways in LOTOS depending upon how the behaviour is represented on which the activity is based. For example, a chain might simply be represented by a sequential ordering of actions within an activity. Alternatively a chain may be modelled through the use of composition operators such as enabling, disabling, or parallel composition if these were used in describing the behaviour expressions of the activity. Regardless of which approach is taken, a chain corresponds to a trace through the behaviour of the activity under consideration.

Formalisation in Z

Given the model of an activity as represented in section 4.2.5, a chain (*Chain*) may be represented by:

$$Chain == \{sa : \text{seq}\ action \mid (\exists act : activity \bullet (\forall a_1; a_2 : action \mid \langle a_1; a_2 \rangle \in sa \bullet \{a_1; a_2\} \subseteq first\ act \wedge (a_1 \mapsto a_2) \in second\ act))\}$$

Through considering chains, objects having their own separate behaviours can be reasoned about. That is, dividing and joining actions can be modelled.

4.4.2 Joining Action

A joining action is an action that is shared between two or more chains resulting in a single chain.

Formalisation in LOTOS

A joining action may be represented in three main ways in LOTOS. It is possible to model joining through the successful synchronisation of event offers at a common gate. This requires that one of the event offers should precede **stop** and the event offers preceding these offers were composed through interleaving. We illustrate this with an example. Consider the following LOTOS fragment:

```
process P[a,b,c,d,e]:noexit:=
   a; (b;c;stop |[c]| d;c;e;stop)
endproc
```

Here the action **a** represents a forking action (see section 4.4.3) since it enables two chains. These two chains consist of the action sequences **b,d,c,e** and **d,b,c,e**. Following the synchronisation at gate **c**, one chain stops and the other continues. This results in the behaviour following action **c** being identical for the two chains which can be interpreted as a single chain.

An alternative model of joining can be achieved through the successful termination *exit* of processes. Consider the following LOTOS fragment:

```
process P[a,b,c,d,e,f]:noexit:=
   a;(b;c; exit ||| d;e; exit) >> f; stop
endproc
```

Here the two behaviour expressions that are interleaved with one another join on the successful termination (¡). These two approaches allow for the natural joining of chains, where natural here implies that the chains complete as specified. In LOTOS it is also possible to limit interleaved behaviours through the disabling operator. This can be used to join (terminate) numerous chains simultaneously. Consider the following LOTOS fragment:

```
process P[a,b,c,d,e,f]:noexit:=
   a;(b;c; exit ||| d;e; exit) [> f; stop
endproc
```

Here **f** can be used to stop numerous chains simultaneously. Various combinations of joining chains can be achieved depending on how the disabling operator and the other modelling styles given previously are used in particular behaviour expressions.

Formalisation in Z

A joining action (*Join*) may be represented in Z by an action that takes two chains and produces a single chain.

JoinAction

$$Join : Chain \times Chain \rightarrow Chain$$

$$\forall c_1; c_2; c_3 : Chain \mid c_1 \neq c_2 \neq c_3 \bullet Join(c_1; c_2) = c_3 \Rightarrow (\exists a : action \bullet \langle a \rangle \in c_1 \wedge \langle a \rangle \in c_2 \wedge last\ c_1 \neq last\ c_2 \wedge (a = last\ c_1 \Rightarrow c_3 = tail\ (SeqRestrict(a; c_2))) \vee (a = last\ c_2 \Rightarrow c_3 = tail\ (SeqRestrict(a; c_1))))$$

Here *SeqRestrict* is a function that takes an action and sequence of actions as arguments and produces a subsequence of the sequence argument. This subsequence is given by the sequence of actions following the action argument.

$$SeqRestrict : action \times {}_{seq}\, action \rightarrow {}_{seq}\, action$$

$$\forall a : action;\ sa_1 : {}_{seq}\, action \bullet$$
$$SeqRestrict(a; sa_1) = \textbf{IF } a = head\ sa_1 \textbf{ THEN } sa_1 \textbf{ ELSE } SeqRestrict(a; tail\ (sa_1))$$

It should be noted here that this recursive definition has no base case since its usage requires that the action is in the sequence as a precondition in *JoinAction*.

Dividing actions are actions that enable two or more chains. There are two cases of dividing action: forking actions and spawning actions.

4.4.3 Forking Action

In forking actions the enabled chains eventually join each other.

Formalisation in LOTOS

Dividing actions generally may be represented in LOTOS by actions that precede behaviour expressions composed through either the interleaving or partial parallel composition operators. Despite this, there does not exist an inherent feature of LOTOS for dealing with forking actions due to the problems of the chains having to join again. It is possible to model forking actions in LOTOS provided certain modelling approaches are adopted. Examples of these approaches include:

- actions that precede behaviour expressions composed through the partial parallel composition operator. These behaviour expressions must synchronise on an event, following which only one possible behaviour is possible. This single behaviour may be obtained in several ways. Either all but one of the behaviour expressions have **stop** after the synchronisation action, or some behaviour expressions offer **stop** after the synchronisation action and the others synchronise on all subsequent events. An example showing a forking action where all but one behaviour expressions offer **stop** after the joining action is given in section 4.4.2.

- actions that precede behaviour expressions composed through interleaving or parallel composition operators. These behaviour expressions themselves should precede the disabling operator.

- actions that precede behaviour expressions composed through the interleaving operator. These behaviour expressions must end with **exit** and be followed by an enabling expression ($\gg$). The action that follows this enabling expression corresponds to the joining action. We illustrate this with a LOTOS fragment.

```
process P[a,b,c,]:noexit:=
  a; (b; exit ||| c; exit) >> d; stop
endproc (* P *)
```

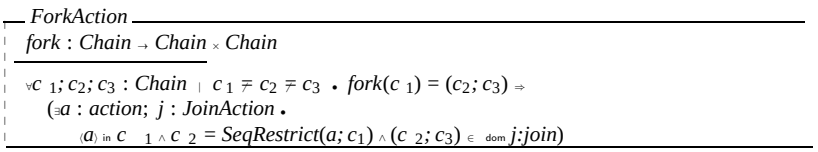
Here **a** represents the forking action and **d** represents the joining action.

It should be noted that combinations of these two approaches are not possible due to the semantics attached to **exit**. Thus (**a;c;stop** $|[c]|$ **d;c;e; exit**) $\gg$ **f; stop** for example is equivalent to (**a;c;stop** $|[c]|$ **d;c;e; exit**) since termination only occurs when all processes are ready to terminate.

As can be seen the modelling of forking actions in LOTOS depends to a great extent on how the the specification is written so as to ensure that the enabled chains join one another.

Formalisation in Z

Given the model of chains in section 4.4.1, a forking action may be represented by a schema that takes a chain and produces two chains. This may be represented by:



4.4.4 Spawn Action

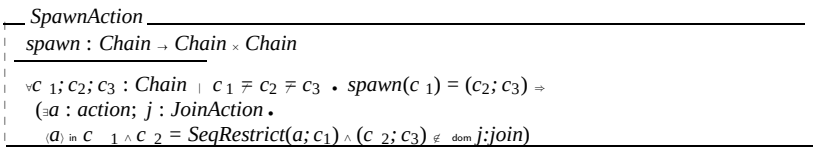
In spawning actions the enabled chains are not required to join one another.

Formalisation in LOTOS

In LOTOS, spawning actions can be modelled in several ways. As with forking actions, spawning actions are represented by actions that precede behaviour expressions combined through the interleaving or partial parallel composition operators. Unlike forking actions, spawning actions do not require that these behaviour expressions join again. As a result spawning actions may be modelled by any action that precedes behaviour expressions combined through the interleaving or partial parallel composition operators, *i.e.* there is no prescription that the chains join again.

Formalisation in Z

Given the model of chains in section 4.4.1, a spawning action may be represented by a schema that takes a chain and produces two chains. This may be represented by:



4.5 Summary

This chapter has argued in detail how an architectural semantics can alleviate many of the problems involved in developing specifications for a particular problem domain. The different approaches possible were presented and their advantages and disadvantages discussed. We then focused on applying LOTOS and Z to develop an architectural semantics for distributed systems based upon the reference architecture of Open Distributed Processing. The approach taken was based upon interpretation.

As was seen, neither LOTOS or Z could formalise all of the concepts. As argued in section 4.1 this is only to be expected. Some concepts are so generic that their formalisation is impossible, *e.g.* type. It was stated in section 4.1 that the lack of a formal interpretation need not imply that the definition is wrong, but simply that the concept does not equate well with the semantics of the formal language. The question should be asked here, whether this is really the case? If a concept is so generic that it results in an infinite number of possible solutions, then is the concept worth keeping in that form?

With regard to the languages themselves, they both have their own advantages and disadvantages for developing an architectural semantics. LOTOS, whilst suited to describing low level behaviours is limited by the fact that it does not offer features inside of the language for reasoning about these behaviours. With regard to developing an architectural semantics this point is crucial. The idea of specifying a library of components that can be used to construct specifications of distributed systems is a highly appealing one, but without some means whereby they can be successfully composed, then the advantages to be gained

from the work are severely limited. The term successful is key here. LOTOS provides numerous object and behavioural composition operators as presented in section 4.3.1. Unfortunately, these operators do not allow any kinds of checks on the behaviours being combined to be made. As a result, the composed behaviours may deadlock, livelock, etc. The significance of this with regard to the development of an architectural semantics are great. Without combinators that deal with behaviour checks, the "plugging together" of the components cannot be guaranteed to have the desired effects.

A less grandiose proposal is thus to deal with the syntactic aspects of the behaviours. Certainly, these are areas that can be checked *a priori*. Unfortunately, LOTOS does not allow the separation of signature from behaviour directly. We show how syntactic reasoning and checking can be achieved in Act One in chapter 7.

Z on the other hand in its classical usage is severely limited by its lack of features for dealing with object-orientation. Whilst not such a great problem for describing single objects, Z is severely limited in describing multi-object systems. As a consequence, a more abstract model of actions was presented in Z. This was then used to describe behaviour in a way that allowed for much more scope in treating objects. This includes composition of objects. We shall see in the following chapters how powerful the abstract approach presented here is. That is, we can reason about composition from many levels, *e.g.* from syntactic aspects to behavioural issues. The abstract model given also allows for the composition of systems based on non-functional aspects of those systems, *i.e.* aspects that the behaviour and signature alone do not capture.

The most important question that could be asked with regard to this chapter is, does this work help specifiers write specifications of ODP systems? Certainly, the formalisation of the concepts provides guidance as to how a concept might best be formalised in a given formal language. This information may then subsequently be used by a specifier. It could be argued that the benefits to the specifier are to some extent limited though. The primary reason for this being the lack of prescription. The main cause of this is the lack of prescriptivity in the ODP-RM itself. This is hardly surprising since ODP is a framework for developing multiple distributed systems and not a single one. Hence, the concepts are necessarily generic. We shall see in the following chapter that ODP becomes more prescriptive in its definition of concepts, and importantly, how they can be composed with one another. This in turn, provides us with more possibilities for developing a more constructive[5] architectural semantics.

[5] In the sense of re-usable specification fragments.

Chapter 5

A More Prescriptive Architectural Semantics

This chapter focuses on developing a more prescriptive architectural semantics for ODP based on formalising the computational viewpoint language in LOTOS and Z. Comparisons are made on the suitability of these languages for this purpose.

We also outline some of the issues involved in the development of an architectural semantics for the other viewpoint languages. We focus in particular on the enterprise and information viewpoint languages.

5.1 The Computational Viewpoint Language of ODP

The computational viewpoint of ODP deals with the functional decomposition of a distributed system. That is, it is from this viewpoint that the system is seen as a collection of objects that interwork to achieve some overall purpose. As such, this viewpoint contains the concepts and rules associated with computational objects and their associated interfaces.

Interfaces in the computational viewpoint consist of a signature, a behaviour specification and an environment contract. An interface signature as discussed in section 4.3.12 represents the syntactic aspects of the functionality (behaviour) found in that interface. The behaviour specification as discussed in section 4.2.2 represents what the effects are of invoking the actions described in the signature. The environment contract represents aspects associated with the interface that its signature and behaviour specification alone do not capture. These might, for example, relate to usage or management constraints. We shall discuss environment contracts and issues in their formal representation in more detail in the following chapter.

The computational viewpoint identifies three particular interface kinds:

- **operational interfaces:** are used to represent classical (RPC-like) interactions. Operational interfaces contain operations which may be either announcements or interrogations. Announcements are used for sending messages where no response is required. Interrogations are used for sending messages where responses (results) are required.

- **stream interfaces:** are used for dealing with continuous flow of data, *e.g.* multimedia. A stream interface consists of a collection of flows of information. The exact nature of these flows is not discussed in ODP. Despite this we show how aspects of multimedia flows can be represented formally in section 5.1.2.

- **signal interfaces:** signal interfaces contain signals, where a signal represents a single, shared atomic interaction between computational objects. Signals are the most basic unit of interaction in the computational viewpoint.

These three kinds of interface have specific rules defined in ODP that relate to how they can be constructed and subsequently composed with one another — or in ODP terminology *bound*. We discuss

these rules in more detail in the following sections when we formalise the computational viewpoint language in LOTOS and Z. Before formalising the specific concepts and structuring rules found in the computational viewpoint, it is first of all necessary for us to provide formal descriptions of the more elementary concepts used there. Specifically, we address those dealing with parameters associated with operational and signal interface signatures and those dealing with the causality of interfaces or, as the case might be, actions contained in those interfaces.

5.1.1 Background Concepts for Computational Viewpoint

Operational and signal interface signatures may have parameters associated with them. These parameters might for example be required as inputs to operations and signals. These parameters may be basic, *e.g.* of type Boolean or integer, or more complex, *e.g.* references to interfaces where complex behaviours might exist. To formalise parameters it is necessary to introduce two concepts: names for things and types for things.

Formalisation in LOTOS

Names are simply labels. As we shall see, the computational viewpoint requires that checks, *e.g.* for equality, are done on these labels when interfaces are constructed. We may represent names generally by:

```
type Name is Boolean
     sorts Name
     opns newName: -> Name
          anotherName: Name -> Name
          _eq_,_ne_: Name, Name -> Bool
endtype (* Name *)
```

For brevity sake we omit the equations, which are expected to be obvious. It is possible to be more prescriptive here, *e.g.* using character strings from the LOTOS library. The only thing we are interested in regarding names is that we can determine their equality or inequality.

As discussed in chapter 4, a type in the ODP sense may not be interpreted directly in the process algebra part of LOTOS. It is however possible to model types through the Act One part of LOTOS. Unfortunately, whilst Act One was designed specifically for representing types, it is limited in the ways in which types and types relationships are checked. For example, it is not possible to check subtyping or equivalence up to isomorphism between types due to type equality in Act One being based on name equivalence of sorts. We discuss this and other related issues in more detail in chapter 7. As a basis for reasoning here we introduce an elementary notion of types that allows us to test for equality, inequality and subtyping.

```
type AnyType is  Boolean
     sorts AnyType
     opns newType: -> AnyType
          anotherType: AnyType -> AnyType
          _eq_,_ne_,_isSubtype_: AnyType, AnyType -> Bool
endtype (* AnyType *)
```

The equations that enable subtyping and hence type equality and inequality are given in chapter 7.

A parameter is a relation between a name and its underlying type representation. Thus a parameter may be represented by:

```
type Param is Name, AnyType
     sorts Param
     opns newParam: Name, AnyType -> Param
          _eq_,_ne_,_isSubtype_: Param, Param -> Bool
endtype (* Param *)
```

As previously, we require checks on the equality or inequality of parameters as well as when one parameter is a subtype of another. Two parameters are in a subtype relationship when their types are in a subtype relationship.

It is also useful for us to introduce sequences of these parameters. The introduction of sequences of parameters, as opposed to sets of parameters say, is not arbitrary. Having sequences of parameters allows us to overcome peculiarities found in the computational viewpoint related to type checking. These are discussed in more detail in section 5.1.2.

```
type PList is String actualizedby Param
     using sortnames PList for String Param for Element Bool for FBool
     opns _isSubtype_: PList, PList -> Bool
endtype (* PList *)
```

Here we use the type *String* from the LOTOS library actualised with the type *Param* defined previously. We also include an operation here *isSubtype* that can check whether one sequence of parameters is a subtype of another. One parameter list is a subtype of a second when all of the parameters it contains are subtypes of those found in the first. In addition the parameters should be in the same position in their respective lists.

Computational interfaces may also have causalities associated with them. A causality may be regarded as an implicit assumption on the expected behaviour of the interface — clients request services to be performed by servers who offer those services. As discussed in section 4.2.1 there does not exist the possibility to represent causality in a formal way in LOTOS. Causality may only informally be attached to event offers (behaviours). This can be achieved through explicitly labelling the event offer or through interpreting the synchronisation form, *e.g.* value passing, as a causal relationship. Explicit labelling of events with causality labels was also done in [203].

Consideration of including a label of causality in event offers is required though. If we include a label of causality in an event offer then this will influence the synchronisation possibilities in LOTOS. For example, the event offers *g !client;*$_{||}$*g !server;* will deadlock. An alternative approach is to model these event offers as *g !client;*$_{||}$*g ?c: Causality*. Unfortunately, this model of causality is not compatible with the other informal approach to modelling causality in LOTOS, *i.e.* causality based on value passing. It is not normally the case that we pass a label of causality as a value as we might a parameter say. As a result we consider causality here only as value passing and we do not attempt to deal with specific labels for causality.

Formalisation in Z

Given that we are not interested in how names for things such as operations and parameters are structured, we may represent names (*Name*) in Z simply as a basic data type. There are also likely to be a large number of different types in the system, *e.g.* action types, object types, interface types, which are likely to have different type representations. We introduce the basic type (*AnyType*) to represent the set of all types that exist in the system.

As above, the parameters that are associated with interfaces to computational objects consist of a name and a type. It should always be possible to determine the type of a parameter in a given system. Hence we introduce *Parameter* as an injective function from names to types.

$$Parameter : Name \rightarrowtail AnyType$$

It is also useful to introduce sequences of these parameters.

$$PList == \operatorname{seq} Parameter$$

As discussed in section 4.2.1 it is not the case that causality may be modelled formally in the Z language. Cause and effect relationships may only be considered informally. One way of achieving this is to model causality as a free type.

$$Causality ::= Producer \mid Consumer \mid Initiator \mid Responder \mid Client \mid Server$$

We shall see that Z in its classical usage suffers from the same problems as LOTOS when causality is introduced, *i.e.* the label can cause problems in meaningful interactions. Applying Z in a more abstract manner allows such issues to be overcome though.

5.1.2 Formalising Computational Interface Signatures

As discussed the computational viewpoint identifies operational, signal and stream interfaces. Each of these has a syntactic element to them, *i.e.* its signature. ODP provides specific rules on how these interface signatures may be constructed with regard to naming considerations. Once constructed, these naming considerations are used to check on whether two interfaces can interact successfully, or be replaced for one another. We consider now how these signatures may be represented in LOTOS and Z.

Formalising Operational Interface Signatures in LOTOS

As discussed above, operational interfaces consist of operations which can be interrogations or announcements. Interrogations consist of an invocation action followed by a non-empty finite set of termination actions. Announcements consist of only an invocation action.

An invocation action consists of a name for the invocation and the number, name and type of the argument parameters associated with the invocation. Simplistically, we might represent a client announcement invocation by the following LOTOS fragment:

```
<g> !<invName> !<inArg1> !<inArg2> ...!<inArgn>;
```

Here we adopt the notation that $< X >$ represents a placeholder for an X, *i.e.* g, *invName* and $inArg_i$ represent placeholders for the gate, the name of the invocation and the parameters that are associated with the invocation respectively. The problem with this approach for developing an architectural semantics is that it leads to immediate deadlocks with processes that do not have similar numbers of parameters. Since we do not know how many parameters event offers have associated with them, a more flexible approach for representing client invocations is:

```
<g> !<invName> !<inArgs>;
```

Here *inArgs* is the list of arguments (as a single structured value) associated with the invocation. Through this structure, processes need not necessarily deadlock immediately if the parameters are not compatible. Server announcement invocations may be represented as:

```
<g> ?<invName: Name> ?<inArgs: PList>;
```

Announcement invocations and invocations for interrogations differ in that the latter expects results to be returned. The types of the possible results are passed in the invocation and the values of the results are sent in a termination. It is of course quite possible to model the expected result types in the parameter *inArgs*. Instead we model client invocations for interrogations as:

```
<g> !<invName> !<inArgs> !<outArgs>;
```

The invocations associated with server interrogations may be represented as:

```
<g> ?<invName: Name> ?<inArgs: PList> ?<outArgs: PList>;
```

These event offer structures adequately capture the text given in ODP with regard to client invocation structures. It is normally the case that object based systems require objects to be referenced in communications however. This approach does not allow objects to be referenced. To overcome this most object based LOTOS specification styles, *e.g.* [44, 147], include a field in the event structure that denotes the object the message is intended for.

The notion of object identity does not play an especially important role in the computational viewpoint though. It is predominantly the interfaces to objects that are of more concern. The information required to find and subsequently access an interface is contained within an interface reference. This information might for example include: the location of the interface; the services the interface offers; constraints on how the interface is accessed, *e.g.* security considerations. Currently interface references and binding in ODP are the subject of standardisation [113].

Whilst it is quite possible to model an interface in the process algebra (see section 4.2.4), it is not possible to model a reference to that interface in the process algebra that, loosely speaking, captures the

functionality of that interface. Object identifiers as found in Moreira's work [147] for example, allow structured interactions between objects (processes) to be achieved. It is not the case though that a process can obtain a reference for an object and be able to determine *a priori* that the reference denotes an object with certain behaviour. That is, object identities support references to objects but these references do not contain sufficient information to restrict potentially unwanted behaviours between objects.

To overcome this, we model interface references in Act One. Given that an interface reference captures, amongst other things, the signature of the interface, we provide an Act One model of signatures for operations. Operations consist of a name, a sequence of inputs and possibly a sequence of outputs[1]. This may be represented by the following LOTOS fragment:

```
type Op is Name, PList
     sorts Op
     opns makeOp: Name, PList -> Op
          makeOp: Name, PList, PList -> Op
          getName: Op -> Name
          getInps: Op -> PList
          getOuts: Op -> PList
     eqns forall n: Name; pl1, pl2: PList
          ofsort Name
             getName(makeOp(n,pl1,pl2)) = n;
          ofsort PList
             getInps(makeOp(n,pl1)) = pl1;
             getInps(makeOp(n,pl1,pl2)) = pl1;
             getOuts(makeOp(n,pl1)) = <>;
             getOuts(makeOp(n,pl1,pl2)) = pl2;
endtype (* Op *)
```

We note here that we model two forms of operations: those that do not expect results and those that do expect results. We shall discuss the form of this modelling of operations in more detail in chapter 7 and how it can be used for determining structural relationships, *e.g.* subtyping, between signatures. We also introduce sets of these operations:

```
type OpSet is Set actualizedby Op
     using sortnames OpSet for Set
                     Op for Element
                     Bool for FBool
endtype (* OpSet *)
```

Now a *direct* interface reference may be represented by the following LOTOS fragment:

```
type IRef is Location, OpSet, Constraints
     sorts IRef
     opns makeIRef : Location, OpSet, Constraints -> IRef
          NULL     : -> IRef
          getLoc   : IRef -> Location
          getOps   : IRef -> OpSet
          getCon   : IRef -> Constraints
          _eq_,_ne_: IRef, IRef -> Bool
     eqns forall l: Location; o: OpSet; c: Constraints; ir1, ir2: IRef
          ofsort Location
             getLoc(makeIRef(l,o,c)) = l;
          ofsort OpSet
             getOps(makeIRef(l,o,c)) = o;
          ofsort Constraints
             getCon(makeIRef(l,o,c)) = c;
          ofsort Bool
             ir1 eq ir2 = (getLoc(ir1) eq getLoc(ir2)) and
                          (getOps(ir1) eq getOps(ir2)) and
                          (getCon(ir1) eq getCon(ir2));
endtype (* IRef *)
```

[1] For simplicity sake we do not consider here whether the operation is of infix, prefix or suffix notation. This is considered in more detail in chapter 7

Here *Location* is a type used to determine the location, *e.g.* network address, at which the interface exists. For simplicity sake we treat this as a basic type (not given here) that can check the inequality or inequality of values modelling locations. In reality, locations may be more complex, *e.g.* they might be modelled as sets of context-specific information about where the interface can be found. For similar reasons we also do not deal with the possible relocation of objects and how this manifests itself in the interface reference. This might for example be achieved through an extra field in the interface reference that denotes a location where an object exists that keeps track of locations and relocations of interfaces. *Constraints* represents other information associated with the interface, *e.g.* information related to expected quality of service or interface usage constraints. We investigate *Constraints* in more detail in the following chapter.

Here we note that equality of interface references is based on the complete information in the interface reference as opposed to any particular field in the interface reference. The reason is that establishing the identity of elements in a distributed system is a non-trivial task, *e.g.* where different naming domains exist. We also note that it is possible to have *indirect* interface references also [113]. For example, interface references may be passed between systems and domains that do not understand the format of the interface references, *e.g.* different IDLs are used to specify the signatures of the services on offer. In this case, opaque information is given along with a reference to an interpreter that can translate the opaque information into a format understood in that domain.

We also introduce sets of these interface references.

```
type IRefSet is Set actualizedby IRef
     using sortnames IRefSet for Set
                     IRef for Element
                     Bool for FBool
endtype (* IRefSet *)
```

>From these considerations we may model an operational interface signature for a client through the following process definition.

```
process OpIntSigClient[ g... ](iref: IRef, known: IRefs, ...):noexit:=
  g !<invName> !<SomeIRef> !<inArgs>;                      ...(* other behaviour *)
   []... (* other announcements *)
  []
  g !<invName> !<SomeIRef> !<inArgs> !<outArgs>;           ...(* other behaviour *)
    (g ?<termName: Name> !iref ?<outArgs: PList>;
      [ not(makeOp(termName,outArgs) IsIn getOps(iref))] -> ...(* return error message *)
        []
      [ makeOp(termName,outArgs) IsIn getOps(iref)] ->     ...(* other behaviour *)
    [] ... (* other terminations *))
  [] ... (* other interrogations *)
 endproc (* OpIntSigClient *)
```

Here the comments and use of dots (...) are used to indicate that this specification fragment has to be extended. The dots used in the formal parameter list might be replaced with parameters used to model state information for example.

This specification fragment requires that the process is instantiated with at least one gate which corresponds to the interaction point at which the interface exists. The process should also be instantiated with a set of interface references and its own interface reference. We note here that it is not possible to write predicates on the invocations sent. To do so would require a level of prescriptivity that we do not have, *e.g.* ensuring that *SomeIRef* is an interface reference that exists in the set of known interface references associated with the process. It is possible to perform checks on terminations though, *i.e.* the terminations received should be one of the operations associated with that interface reference.

We also note that we have used the choice operator here to model the composition of individual announcements and interrogations. It is quite possible to use several other composition operators here, *e.g.* interleaving. If interleaving composition is used then multiple invocations can be received before any terminations say, are sent. Since interfaces usually have some form of existence, *i.e.* they offer operations that can be invoked more than one time, the comments representing other behaviours are likely to contain recursive[2] process instantiations. Through using the choice operator we have a form of blocking

[2]But see also the discussion of recursion in LOTOS in section 4.2.5.

of operations, *i.e.* should an invocation associated with an interrogation occur, then a termination has to be issued before any other invocations are accepted. This is not necessarily the case with announcements, *e.g.* they may be followed directly by a process instantiation which allows the other invocations to occur. Similar arguments hold for all other processes representing computational interface signatures.

ODP also requires that the names of invocations (and terminations) in an operational interface signature are distinct in the context of that signature. It is possible to specify the sets of operations associated with an interface reference so that they all have unique names, however enforcing this in the event offers is not possible.

Operational interfaces signatures for servers may be represented by:

```
process OpIntSigServer[ g... ](iref: IRef, known: IRefs, ...):noexit:=
  g ?<invName: Name> !iref ?<inArgs: PList>;
    ([ not(makeOp(invName,inArgs) IsIn getOps(iref))] ->         ...(* ignore/other behaviour *)
     []
     [ makeOp(invName,inArgs) IsIn getOps(iref) ] ->             ...(* other behaviour *)
  []... (* other announcements *))
  []
  (g ?<invName: Name> !iref ?<inArgs:PList> ?<outArgs:PList>;    ...(* other behaviour *)
    ([ not(makeOp(invName,inArgs,outArgs) IsIn getOps(iref))] -> ...(* return error mes-
sage *)
      []
      [ makeOp(invName,inArgs,outArgs) IsIn getOps(iref) ] ->    ...(* other behaviour *)
        g !<termName> !<SomeIref> !resList ;                     ...(* other behaviour *)
        [] ... (* other terminations *))
  [] ... (* other interrogations *))
 endproc (* OpIntSigServer *)
```

As with client interface signatures, a server interface signature has a set of known interface references and a reference for itself. This latter interface reference is used to ensure that the announcement or interrogation invocations the server receives are those that were expected, *i.e.* they were in the set of operations associated with that interface reference. If these invocations were not acceptable, *e.g.* the parameters were not correct or the operation requested was not available, then error handling behaviours are taken. In the case of announcements this might result in a recursive call with the formal parameter list being unchanged. It is also possible to use a guard here to prevent the event from occurring in the first place. We do not do so since this might produce unwanted deadlocks in the specification. In the case of interrogations this would result in some form of error message being returned.

As with client operational interfaces it is possible to require that the messages received are those that were expected. It is not possible to have prescriptions on the messages sent though. It could be argued that this limitation is not necessarily a bad thing. Provided every process treats received messages the same way, then no 'bad things' should ever happen. The worst thing that could happen for a sent message is a reply of 'message not supported by this interface'. It is not the case that a sent message can cause deadlocks through its format not being understood for example.

Formalising Operational Interface Signatures in Z

As discussed above, invocation actions consist of names and the number, name and type of parameters associated with the invocation. This can be represented in Z by the following schema:

InvTemplate

$$invName : Name$$
$$inArgs : PList$$

A termination action as represented by a termination name, and number, names and types of result parameters may be represented in Z by:

TermTemplate

$$termName : Name$$
$$outArgs : PList$$

There are numerous ramifications of this model for invocation and termination templates that we need to discuss. Indeed these ramifications reflect directly onto the entire formalisation of the computational viewpoint language in Z and more, importantly here, on the usage of Z to develop a library of re-usable specification fragments as might be found in an architectural semantics.

Consider for example the representation of an invocation. We might define the client side and server side of an invocation as:

$$SomeInvocation \mathrel{\widehat=} InvTemplate[aName! = invName; aPL! = inArgs] \gg InvTemplate[aName? = invName; aPL? = inArgs]$$

Here *aName*! and *aPL*! represent the actual parameters that are inserted on the client side of the invocation and *aName*? and *aPL*? the actual parameters inserted on the server side. This simple model of a re-usable action template is adequately represented in Z since the action being modelled is itself atomic. This model of an action template does not lend itself to modelling object-oriented systems however. That is, the client side and server side of the action template are represented in the same schema. As stated in section 3.3.5, most object-oriented extensions to Z have overcome this problem through the introduction of a class schema for grouping (and hence separating) operations. With this approach, it is also not possible to model non-atomic actions as might be found in ODP. For example, consider the modelling of an interrogation in Z. As stated, an interrogation is modelled in ODP as an invocation followed by one or more terminations. The client side of an interrogation might be represented by:

$$ClientInterrogation \mathrel{\widehat=} InvTemplate[aName! = invName; aPL! = inArgs] \land (TermTemplate[tName1? = termName; PL1? = outArgs] \lor TermTemplate[tName2? = termName; PL2? = outArgs] \lor \ldots)$$

Here the dots represent possible further terminations. With this model of the client side of an interrogation, the problem is that the schema itself represents a single atomic action. That is, it is not the case that the invocation occurs first and is followed by one or more terminations. The whole *ClientInterrogation* represents a single atomic action template. The problems of atomicity are further compounded when interrogations themselves are modelled. Assuming we model server sides of interrogations as invocations accepting values and terminations producing values, then an interrogation might be modelled as:

$$Interrogation \mathrel{\widehat=} ClientInterrogation \gg ServerInterrogation \gg ClientInterrogation$$

Here there is in reality no ordering of operation schemas as such. *Interrogation* simply represents a schema that has existentially quantified inputs and outputs through the piping operator. Further, these inputs and outputs are not quantified in an ordered fashion, *i.e.* it is not the case that the first client interrogation passes parameters and an invocation name to the server interrogation which then passes information to the client interrogation again. The quantification is made completely on all of the schemas at the same time.

Other approaches to modelling interrogations suffer from similar problems. For example, modelling an interrogation as an invocation and a finite set of terminations might be represented by:

$$Interrogation \mathrel{\widehat=} SomeInvocation \land (SomeTermination_1 \lor SomeTermination_2 \ldots)$$

Here as before *Interrogation* represents a single schema. Thus there is no notion of the invocation occurring first and terminations following afterwards. This direct approach to modelling ODP constructs in Z is also limited by the lack of checks that can be made. For example, there is no notion of type checking here. It is of course possible to model all actions, *i.e.* invocations and terminations, as separate entities that exist in the specification with informal commentary linking them. This is a serious limitation with regard to the usefulness of Z for developing an architectural semantics, *i.e.* being able to specify simplistic actions only results in an almost trivial architectural semantics.

>From this it is obvious that Z when used in its classical style of schemas and schema calculus is limited in modelling any form of complex interaction patterns that might exist between computational objects. As

a result it is necessary to develop a more abstract view of actions and behaviours generally in Z. Hence we use the abstract Z approach developed in chapter 4.

An abstract model of the computational viewpoint requires that we represent the syntactic elements of the actions found in the interface signatures of computational objects. We have seen how invocations and terminations may be represented directly. Announcements as represented as single invocations may be represented in Z by:

$$\begin{array}{|l} \textit{AnnSig} \\ \hline inv : InvTemplate \\ \hline \end{array}$$

Interrogations as invocations followed by a non-empty finite set of terminations may be represented in Z by:

$$\begin{array}{|l} \textit{IntSig} \\ \hline inv : InvTemplate \\ terms : \mathbb{F}_1\, TermTemplate \\ \hline \forall t_1; t_2 : TermTemplate \bullet \\ \quad t_1 \in terms \land t_2 \in terms \land t_1 \neq t_2 \Rightarrow t_1.termName \neq t_2.termName \\ \hline \end{array}$$

Here we also state that the terminations associated with a given invocation should have distinct names.

Operational interface signatures consist of sets of announcements and interrogations, and the interface as a whole is given a causality: client or server. Naming considerations of the components of the interface are required. That is, all invocation names in the interface are required to be unique. All termination names associated with a given invocation are also required to be unique. This can be represented as:

$$\begin{array}{|l} \textit{OpIntSig} \\ \hline anns : \mathbb{F}\, AnnSig \\ ints : \mathbb{F}\, IntSig \\ role : Causality \\ \hline \#anns + \#ints \geq 1 \\ role \in \{Client; Server\} \\ (\forall as_1; as_2 : AnnSig;\ is_1; is_2 : IntSig;\ t_1; t_2 : TermTemplate \bullet \\ \quad as_1 \in anns \land as_2 \in anns \land as_1 \neq as_2 \Rightarrow as_1.inv.invName \neq as_2.inv.invName \land \\ \quad is_1 \in ints \land is_2 \in ints \land is_1 \neq is_2 \Rightarrow is_1.inv.invName \neq is_2.inv.invName \land \\ \quad is_1 \in ints \land as_1 \in anns \Rightarrow is_1.inv.invName \neq as_1.inv.invName) \\ \hline \end{array}$$

Here we also state that the invocation names of all announcements and interrogations in a given signature are distinct. We also require that there exists at least one announcement or interrogation signature in the interface signature.

It should be noted that ODP also states that all of the parameter names associated with invocations and terminations are also required to be unique. This could be represented by further predicates, namely:

$$\begin{array}{l} (as_1 \in anns \land (p_1) \in as_1.inv.inArgs \land (p_2) \in as_1.inv.inArgs) \forall \\ (is_1 \in ints \land (p_1) \in is_1.inv.inArgs \land (p_2) \in is_1.inv.inArgs) \forall \\ (is_1 \in ints \land t_1 \in is_1.terms \land (p_1) \in t_1.outArgs \land (p_2) \in t_1.outArgs) \\ \qquad \land p_1 \neq p_2) \Rightarrow first\ p_1 \neq first\ p_2) \end{array}$$

Little justification is given in ODP as to why parameter names should be distinct. One possible reason [95] is that they allow direct correspondences between signatures to be established. Modelling the parameters associated with invocations and terminations as sequences overcomes this problem. Corresponding signatures then require that parameters have the same positioning in their respective sequences.

Formalising Signal Interface Signatures in LOTOS

As stated previously, signals are atomic actions. They result in a one-way communication from an initiating to a responding object. Structurally, a signal signature is similar to an invocation for an announcement (or a termination associated with an interrogation) as given previously, *i.e.* it consists of a name (for the signal), a sequence of parameters associated with the signal and an indication of causality. As done previously, we represent this indication of causality implicitly through the event offer. Further, since all events in LOTOS are atomic, there is no inherent distinction between events modelled as announcements or signals.

Signal interface signatures differ from operational interface signatures though in that they do not require that the interface as a whole is given a causality. Instead, signal interface signatures may contain signals with either initiating or responding causalities. From this we may model a signal interface signature in LOTOS by:

```
process SignalIntSig[ g... ](iref: IRef, known: IRefs...):noexit:=
  g !<sigName> !<SomeIRef> !<pl>;                        ...(* other behaviour *)
  []... (* other initiating actions *)
  []
  g ?<sigName: Name> !iref ?<inArgs: PList>;
   ([ not(makeOp(sigName,inArgs) IsIn getOps(iref))] -> ...(* unsuccessful behaviour *)
     []
     [ makeOp(sigName,inArgs) IsIn getOps(iref) ] ->     ...(* successful behaviour *))
  []... (* other responding actions *)
endproc (* SignalIntSig *)
```

Here we state that a signal interface consists of a collection of event offers. These event offers may model either outgoing signals, *i.e.* those event offers with *!* prefixing the signal name and list of parameters, or incoming signals, *i.e.* those event offers with *?* prefixing the signal name and list of parameters. In the case of incoming signals, it is possible to check that the incoming signal is one expected, *i.e.* the signal is in the set of allowed signals associated with that interface reference.

As with operational interface signatures, ODP requires that the names of signals in a signal interface signature are distinct in the context of that signature. It is possible to specify the sets of operations associated with an interface reference so that they all have unique names, however, enforcing this in the event offers is not possible.

Formalising Signal Interface Signatures in Z

As stated, signals represent the most basic unit of interaction in the computational viewpoint. They have associated with them a name, the number, names and types of parameters, and a causality which can be initiating or responding. A signal signature may thus be represented by:

$$
\begin{array}{|l}
\textit{SignalSig} \\
\hline
\textit{signalName} : \textit{Name} \\
\textit{args} : \textit{PList} \\
\textit{role} : \textit{Causality} \\
\hline
\textit{role} \in \{\textit{Initiator}; \textit{Responder}\} \\
\hline
\end{array}
$$

Unlike operational interface signatures where causalities are attached to interfaces as a whole, causalities are attached to individual signals. A signal interface signature consists of a set of signal signatures. This can be represented in Z by:

$$
\begin{array}{|l}
\textit{SigIntSig} \\
\hline
\textit{sigs} : \mathbb{F}_1 \textit{SignalSig} \\
\hline
\forall ss_1, ss_2 : \textit{SignalSig} \bullet ss_1 \in \textit{sigs} \land ss_2 \in \textit{sigs} \land ss_1 \neq ss_2 \Rightarrow ss_1.\textit{signalName} \neq ss_2.\textit{signalName} \\
\hline
\end{array}
$$

Each signal name associated with a given signal interface signature is required to be unique. We note here that ODP also requires parameter names associated with signals to be unique also. As with operational interfaces we avoid this through modelling the parameters associated with a signal as a sequence. There should exist at least one signal signature in the signal interface signature.

Formalising Stream Interface Signatures in LOTOS

As discussed, the computational viewpoint also considers interfaces concerned with the continuous flow of data, *e.g.* multimedia. These interfaces are termed stream interfaces. Stream interfaces contain finite sets of flows. These flows may be from the interface (produced) or to the interface (consumed). Each flow is modelled through an action template. Each action template contains the name of the flow, the type of the flow, and an indication of causality for the flow.

The ODP-RM abstracts away from the contents of the flow of information itself. We consider here a generic idea of information flow where the flow of information is represented by a sequence of *frames*. A frame may be regarded as a particular item in the flow of information. We note here that it could be argued that we are taking an engineering as opposed to a computational viewpoint when considering flows of information. ODP abstracts from the nature of the information flow. Flows are regarded in ODP as continous actions. With our approach we model streams as sequences of discrete timed events. On the one hand this allows us to deal with the timing issues of information flows but we achieve this at the cost of losing the continous nature of the flows. Our approach models streams as they might be represented with signals as given in ODP.

Each frame in an information flow can be considered as a unit consisting of data (this may be compressed) which we represent by *Data*[3] and a time stamp used for modelling the time at which this particular frame was sent or received. It is also often the case in multimedia flows that particular frames are required for synchronisation, *e.g.* synchronisation of audio with video for example. Therefore we associate a particular *Name* with each frame. This can then be used for selecting a particular frame from the flow as required. From this, we may model a frame as:

```
type Frame is Name, NaturalNumber, Data, Param
     sorts Frame
     opns makeFrame: Data, Nat, Name -> Frame
          nullFrame: -> Frame
          getData: Frame -> Data
          getTime: Frame -> Nat
          getName: Frame -> Name
          toParam: Frame -> Param
          setTime: Nat, Frame -> Frame
     eqns forall d: Data, s,t: Nat, n: Name
          ofsort Data
           getData(makeFrame(d,t,n)) = d;
          ofsort Nat
           getTime(makeFrame(d,t,n)) = t;
          ofsort Name
           getName(makeFrame(d,t,n)) = n;
          ofsort Frame
           setTime(s,makeFrame(d,t,n)) = makeFrame(d,s,n);
endtype (* Frame *)
```

It should be noted here that we model time as a natural number as done by [60]. It might well be the case that real (dense) time could be used as in [121], or time intervals [47]. For simplicity here though, we restrict ourselves to discrete time represented as a natural number. We also introduce an operation that converts a frame into a parameter. For simplicity we omit the associated equations.

We also introduce sequences of these frames:

```
type FrameSeq is Frame
       sorts FrameSeq
       opns makeFrameSeq: -> FrameSeq
            addFrame: Frame, FrameSeq -> FrameSeq
            remFrame: Frame, FrameSeq -> FrameSeq
            getFrame: Name, FrameSeq -> Frame
            timeDiff: Frame, Frame -> Nat
       eqns forall f1, f2: Frame, fs: FrameSeq, n1,n2: Name
            ofsort FrameSeq
              getTime(f1) le getTime(f2) =>
               addFrame(f1,addFrame(f2,makeFrameSeq)) = addFrame(f2,makeFrameSeq);
            ofsort Frame
              getFrame(n1,makeFrameSeq) = nullFrame;
              n1 ne n2 =>
               getFrame(n1,addFrame(makeFrame(d,t,n2),fs)) = getFrame(n1,fs);
              n1 eq n2 =>
               getFrame(n1,addFrame(makeFrame(d,t,n2),fs)) = makeFrame(d,t,n2);
endtype (* FrameSeq *)
```

[3]It is very likely that *Data* would be modelled through the information viewpoint language. This model might include how the information was compressed, what information was compressed, etc. As such it is not considered further here.

For brevity here we do not supply all of the equations. Frames are added to the sequence provided they have increasing timestamps. An operation is provided for traversing a sequence of frames to find a named frame. We also introduce an operation to get the time difference between time stamps of two frames. It is possible using this operation to specify, for example, that all frames in a sequence are separated by equal time stamps. In this case we have an isochronous flow. We also introduce sets of these sequences of frames:

```
type FrameSeqSet is Set actualizedby FrameSeq
     using sortnames FrameSeqSet for Set FrameSeq for Element Bool for FBool
endtype (* FrameSeqSet *)
```

>From this model of sequences of frames, a stream interface signature may be represented by the following LOTOS fragment:

```
process StreamIntSig[ g... ](iref: IRef, known: IRefSet, fss: FrameSeqSet...):noexit:=
    ConsumeAction[ g...](iref, known, recFrames...) []... (* other consume actions *)
    []
    ProduceAction[ g...](iref, known, FramestoSend, ...) []... (* other produce actions *)
endproc (* StreamIntSig *)
```

As with signal interfaces the notion of causality is applied to individual action templates in the stream interface signature. A stream interface signature contains sets of flows consuming or producing actions. Each flow signature is represented by a process. These processes contain the reference to the stream interface with which they are associated, a set of interface references representing the interface references known to that interface and a sequence of frames to send (in the case of producing flows) or receive (in the case of consuming flows). For brevity we do not specify how the set of sequences of frames that are passed to a stream interface signature are assigned to the producing flows in that interface. When instantiated all consume flows are of course empty. A producing flow may be represented by the following LOTOS fragment:

```
process ProduceAction[ g...](iref: IRef, known: IRefSet, toSend: FrameSeq...)
:noexit:=
  [ toSend ne makeFrameSeq ] -> g !<flowName> !<SomeIRef> !<head(toSend)>;
         ...(* other behaviour and recurse with frame removed from toSend *)
endproc (* ProduceAction *)
```

The process representing a producing flow should have a formal parameter representing the sequence of frames that are to be sent. It might be considered that two sequences of frames would be a better model, *e.g.* one sequence representing the frames to be sent and the other representing the frames sent. This might be used to retransmit lost or delayed frames for example. It is more often the case though that in multimedia flows of information, delayed or lost frames are simply dropped or ignored by the flow consumer. That is, a certain amount of dropped frames is less detrimental to the flow presentation than stopping the flow whilst frames are resent.

There are various possibilities for how the frames get time stamped. We illustrate three. It might be the case that all processes have access to a global clock from which they can establish the current time. The frames sent from a producer could then be timestamped with this time value. The modelling of global clocks in LOTOS is not without its problems however. For example, one model of a clock might be:

```
process Clock[ t ](tnow: Nat) :noexit:=
 t !tnow; Clock[t](tnow) [] i; Clock[ t ](succ(tnow))
endproc (* Clock *)
```

Here the clock either outputs the current time or an internal event occurs and the time is incremented. This model of a global clock may also be found in [44]. This model of time is limited when modelling flows of information. Here, the time itself is based on non-deterministic internal events. Flows of information have explicit temporal requirements that need to be satisfied. As such this model of time is not sufficiently expressive. Modifying this clock process so that the non-determinism is removed, *e.g.* each time the clock is referenced the time is given and then incremented, is also an unsatisfactory model since it can adversely influence concurrent behaviours. That is, LOTOS models concurrent behaviours through the interleaving operator. Concurrency loosely implies that they can happen at the same time. If processes

are time dependent though then this is not the case, *i.e.* one must happen after the other if they access the global clock.

A second approach is to model time in the formal parameter list associated with a process modelling a flow. This might for example be represented by:

```
process ProduceAction[ g,s...](... toSend: FrameSeq, tnow: Nat ...)
:noexit:=
 g !<flowName> !<SomeIRef> !<setTime(tnow,head(toSend))>;
  (* other behaviour and recurse with frame removed from toSend and time incremented *)
 []
 s ?tupdate: Nat;
 ProduceAction[ g,s...](... toSend,tupdate)
endproc (* ProduceAction *)
```

Here we have a local model of time. That is, the process itself keeps track of its current time. With this model of time it is possible to model processes running at different speeds say, *e.g.* where different *tnow* variables and their modifications exist in different processes. If this approach is taken then there should exist some means whereby the current local time can be set, *e.g.* so that processes can re-align their clock values (*tnow* variables). We introduce the gate *s* for this purpose. The current time (local to the process) is time stamped onto the frame being sent through the operation *setTime*. The value that the time variable *tnow* is increased by is proportionate to the rate of the flow. If the modification to the time variable is the same each time a frame is sent then we have an isochronous flow.

It is often the case that levels of control are required for modifying flows of information, *e.g.* send faster or slower as the case might be. In this model, the control is given by the recursive call and how the *tnow* variable is incremented. As such the flexibility of manipulating *tnow* is limited. That is, the operations given in the recursive call are static. Further, if *tnow* represents the local time, then operations to manipulate how time progresses might also seem unnatural. To overcome this, it is possible to model the rate of flow of frames to be sent as a formal parameter. This might be represented by:

```
process ProduceAction[ g, m...](... toSend: FrameSeq, tnow: Nat, rate: Nat ...)
:noexit:=
 g !<flowName> !<SomeIRef> !<SetTime(tnow+rate,head(toSend))>;
          ...(* other behaviour and recurse with frame removed from toSend *)
 [] m ?newRate: Nat [newRate gt 0];
          ...(* other behaviour and recurse with new rate set *)
 [] s ?tupdate: Nat;
          ...(* other behaviour and recurse with new current time set *)
endproc (* ProduceAction *)
```

Here we note that we model the rate as a natural number. This allows the rate of flow to be sped up or slowed down depending on whether the rate is decreased or increased respectively. When instantiated, predicates should be given to ensure that the rate is greater than zero. Proposed new rates are checked to ensure that they are greater than zero. Having a zero or negative value for the rate would allow consecutive frames in the sequence to have decreasing or equal time stamp values. This could destroy the temporal integrity of the flow of information, for example where a negative rate was given that was greater than the time difference between two consecutive frames in the flow.

It is likely that the gate *m* will be internal to the object with which the interface is associated. This gate might be used explicitly for management and control purposes. For example, another interface used for controlling the flow of information might receive a message to slow the speed of flow up (or slow it down). This information would then be used to establish the new rate.

This model of local time is limited to a certain extent in that (non-relativistic) time is usually considered to be global.

A third alternative is to model time entirely through Act One. That is, when the sequence of frames is created, the time stamps are given there. For example, a constructor operation (fs_1 say) that returns a sequence of frames might have equations that give the time stamps explicitly, *e.g.*

```
addFrame(makeFrame(d_1,t_1,n_1),... ,addFrame(makeFrame(d_i,t_i,n_i),makeFrameSeq)))
```

Here the time stamps t_i might be represented by explicit natural numbers say. With this model of sequences of frames, the production of a frame might simply be represented by:

```
process ProduceAction[ g...](... toSend: FrameSeq ...)
:noexit:=
 g !<flowName> !<SomeIRef> !<head(toSend)>;
       ...(* other behaviour and recurse with frame removed from toSend *)
endproc (* ProduceAction *)
```

This model of timed frames is the simplest to represent in the process algebra. Unfortunately, the actual sending of the frame itself is given by the occurrence of the process algebra event. As a result, the time stamping achieved in Act One is independent of the event offer occurrence in the process algebra. Another problem with this model of time stamping frames is that anything like a realistic sized sequence of frames would not be well suited to Act One specification. That is, the equations required for an information flow of several thousand irregularly time stamped[4] frames say would be far too verbose to be practicable. It might be the case that only isochronous flows are considered. In this case, the constant value of the difference between time stamps of consecutive frames could be specified directly in the equations associated with the frame sequence.

One major problem with this model of information flows, though, is that it does not lend itself to changing the rate at which frames are sent. It is often the case that produced information flows should be modified, *e.g.* slowed down or speeded up. The modelling of information flows exclusively in Act One and the occurrence of these flows in the process algebra prohibits modifications to the information flow rate.

Consumption of frames typically has different requirements placed upon it. The need to continually monitor the time stamps of the incoming flow of information is of particular importance. Due to the potential spatial separation of producers and consumers of flows of information, there is often a non-negligible time difference between the sending of a frame from a producer to its arrival at the consumer. This time difference is heavily dependent upon the connection between the producer and consumer of the flow. This connection is likely to have a limit on the information that can be passed through it at any given time. The current usage of this connection will thus influence the speed at which information is passed from producer to consumer. A consumer of an information flow may be represented by:

```
process ConsumeAction[ g,s...](iref: IRef, known: IRefSet, recFrames: FrameSeq, tnow: Nat...)
:noexit:=
 g ?<flowName: Name> !iref ?<inFrame: Frame>;
  ([ not(makeOp(flowName,isParam(inFrame)) IsIn getOps(iref))] -> ...(* unsuccessful be-
haviour *)
   []
   [ (makeOp(flowName,isParam(inFrame)) IsIn getOps(iref)) and
     ((getTime(inFrame) - tnow) gt limit) ] ->                    ...(* frame too late be-
haviour *)
   []
   [ (makeOp(flowName,isParam(inFrame)) IsIn getOps(iref)) and
     ((getTime(inFrame) - tnow) le limit) ] ->
             (* other behaviour, e.g. display frame and recurse with time incremented *)
              (* or recurse with frame added to received frames and time incremented *)
  []
  s !tnow;
  ConsumeAction[ g...](iref,known,recFrames,tnow)
endproc (* ConsumeAction *)
```

Here we reject the frame sent if the flow name and frame are not associated with this consume action. If the flow name and frame are acceptable then we check that the time of the incoming frame is within some limit. We focus in more detail on this limit and other timeliness considerations in the following chapter. If the frame arrives too late then some appropriate behaviour is taken, *e.g.* the frame is dropped, and a recursive call is made with the clock incremented by some amount. If the frame is acceptable to this interface and it does not violate any timing constraints, then either it is displayed or appended to those already received, or possibly a combination of these.

We also include an event offer here that allows the current time of this process to be passed as a parameter to synchronise clocks for example.

[4] *i.e.* consecutive frames in the sequence may have time stamps whose differences are not constant.

Formalising Stream Interface Signatures in Z

As with LOTOS, we model the flow of information in Z in a generic manner. We consider data that can be labelled and that has a temporal ordering given by timestamps. We introduce the basic data type *Data* to model the information itself. A generic frame may be represented by:

Frame

$data : Data$

$timestamp : \mathbb{N}$

$label : Name$

It is possible here to model sequences of frames as with LOTOS and to provide operations for adding or obtaining a frame from a sequence. Instead we show how Z can be used to capture different types of flow characteristic. Flows can be *isochronous* which implies that each frame is sent/received in equal time segments. Alternatively, flows can be *bursty* in nature which implies that the time intervals between successive frames are not necessarily equal. We may thus represent a flow characteristic as:

$$FlowChar ::= Isoch\langle\mathbb{N}\rangle \mid Bursty$$

>From this, we may represent a generic multimedia flow as:

mmFlowType

$frames : \operatorname{seq} Frame$

$flowChar : FlowChar$

$rate : \mathbb{N}$

$\forall f_1; f_2 : Frame \mid \langle f_1; f_2\rangle \in frames \bullet f_2.timestamp > f_1.timestamp \wedge$

$flowChar = Isoch(rate) \Rightarrow$

$(\forall f_1; f_2 : Frame \mid \langle f_1; f_2\rangle \in frames \bullet f_2.timestamp - f_1.timestamp = rate)$

$flowChar = Bursty \Rightarrow$

$(\exists f_1; f_2; f_3 : Frame \mid \langle f_1; f_2\rangle \in frames \wedge \langle f_2; f_3\rangle \in frames \bullet$

$f_2.timestamp - f_1.timestamp \neq f_3.timestamp - f_2.timestamp)$

This states that a multimedia flow type is given by a sequence of frames with some flow characteristic and temporal ordering. All frames in the sequence have time stamps in ascending order. Isochronous flows have frames separated by equal time intervals, whereas bursty flows may have frames separated by unequal time intervals.

A flow signature represented as a name, flow type and causality may thus be represented by:

FlowSig

$fName : Name$

$fType : mmFlowType$

$role : Causality$

$role \in \{Producer; Consumer\}$

Stream interfaces consist of sets of flow signatures. Each flow signature name in a given stream interface signature is required to be uniquely identified. This can be represented as:

StrIntSig

$flows : \mathbb{F}_1 FlowSig$

$\forall fs_1; fs_2 : FlowSig \bullet$

$fs_1 \in flows \wedge fs_2 \in flows \wedge fs_1 \neq fs_2 \Rightarrow fs_1.fName \neq fs_2.fName$

This schema describes the syntactic structure of stream interface signatures satisfying the rules given in the ODP-RM. However, it does not prescribe any particular behaviour. Indeed whilst the LOTOS specification fragments given here have explicit comments that denote areas that require behaviour to be inserted, the Z texts capture only the syntactic elements of computational interfaces. With regard to the formalisation of the ODP architecture, this is all that has been prescribed regarding the action structures. Interfaces have behaviour so we consider now how these syntactic elements can be related to behaviours.

Introducing Behavioural Considerations in Z

As presented in section 4.2.2, behaviour in Z may be represented by a relation between actions. For a pair of actions to be in the relation the postcondition of the first action must satisfy the precondition of the second action. As defined in section 4.2.1, actions may be represented by functions that require a signature for the action and the states that it relates. As discussed in section 4.2.3, it is possible to represent states at many levels of abstraction depending on the way the information in the system is represented. For simplicity here, we consider the most elementary model of a state, *i.e.* as a basic type (*State*). In section 4.2.1 we labelled the syntactic aspect of actions as *ObsActSig* and *IntActSig* depending on their observability. In this chapter we have developed specific action signatures that may be found in interface signatures, namely: *InvTemplate; TermTemplate; SignalSig* and *FlowSig*. We also introduce the action signature *Internal* to represent the structure of internal actions. We may thus represent actions generally by:

$$
\begin{aligned}
action ::= {} & isInvAct\langle InvTemplate \times State \times State\rangle \mid \\
& isTermAct\langle TermTemplate \times State \times State\rangle \mid \\
& isSigAct\langle SignalSig \times State \times State\rangle \mid \\
& isFlowAct\langle FlowSig \times State \times State\rangle \mid \\
& isIntAct\langle Internal \times State \times State\rangle
\end{aligned}
$$

Here we state that the actions in the computational viewpoint may be invocations, terminations, signals, flows or internal. Each of these actions takes a syntactic element and two states, the state required for the action to occur and the state after the occurrence of the action.

Objects and the interfaces that are associated with them are typically not just collections of actions though. As discussed in 4.3.11, typically these actions have some form of ordering relation imposed on them. This ordering is based on the preconditions and postconditions of the actions as shown in section 4.2.2. Objects also remember the effect of actions. As shown in section 4.3.11, we had different ways in which the history of an object could be modelled. For simplicity here we deal with a history as a sequence of states.

We may now represent basic interface templates as schemas consisting of signatures, behaviours and histories where the actions in the behaviour specification relate states in the history. We shall see in the following chapter why we regard these interfaces as basic.

Modelling Operation Interface Templates in Z

An operational interface template may thus be represented by:

OpIntTemplate

$$
\begin{aligned}
& ops : OpIntSig \\
& opbs : behspec \\
& ophis : history \\
\hline
& \forall is : IntSig;\ as : AnnSig;\ s_1; s_2 : State;\ ss : \operatorname{seq} State \mid \\
& \quad is \in ops{:}ints \wedge as \in ops{:}anns \wedge ss \frown \langle s_1; s_2\rangle \operatorname{prefix} ophis \bullet \\
& (\textbf{let}\ ivs == \{it : InvTemplate \mid (it = is{:}inv \vee it = as{:}inv) \wedge \\
& \qquad isInvAct(it; s_1; s_2) \in \operatorname{dom} opbs \cup \operatorname{ran} opbs \bullet isInvAct(it; s_1; s_2)\} \bullet \\
& (\textbf{let}\ tms == \{tt : TermTemplate \mid tt \in is{:}terms \wedge \\
& \qquad isTermAct(tt; s_1; s_2) \in \operatorname{dom} opbs \cup \operatorname{ran} opbs \bullet isTermAct(tt; s_1; s_2)\} \bullet \\
& (\textbf{let}\ ins == \{i : Internal \mid isIntAct(i; s_1; s_2) \in \operatorname{dom} opbs \cup \operatorname{ran} opbs \bullet isIntAct(i; s_1; s_2)\} \bullet \\
& \qquad \langle ivs; tms; ins\rangle \operatorname{partition} \operatorname{dom} opbs \cup \operatorname{ran} opbs)))
\end{aligned}
$$

Here we state that the only actions that can be found in an operation interface template are either invocation actions, termination actions or internal actions. Further, the actions found in the behaviour specification relate states found in the history of states associated with that interface.

Modelling Stream Interface Templates in Z

Stream interface templates are represented similarly, except that the actions found in the interface are either flows or internal. This may be represented by:

StrIntTemplate

$$
\begin{array}{l}
streams : StrIntSig \\
strbs : behspec \\
strhis : history \\
\hline
\forall s_1; s_2 : State;\ ss : \mathrm{seq}\ State \mid ss\ \neg\langle s_1; s_2\rangle \in \mathrm{prefix}\ strhis \bullet \\
\quad (\mathbf{let}\ flowActs == \{fs : FlowSig \mid fs \in streams.flows\} \\
\qquad isFlowAct(fs; s_1; s_2) \in \mathrm{dom}\ strbs \cup \mathrm{ran}\ strbs \bullet isFlowAct(fs; s_1; s_2)\} \bullet \\
\quad (\mathbf{let}\ otherActs == \{ia : Internal \mid isIntAct(ia; s_1; s_2) \in \mathrm{dom}\ strbs \cup \mathrm{ran}\ strbs \bullet \\
\qquad isIntAct(ia; s_1; s_2)\} \bullet \langle flowActs; otherActs\rangle \in \mathrm{partition}\ \mathrm{dom}\ strbs \cup \mathrm{ran}\ strbs))
\end{array}
$$

This states that the only actions that can be found in the behaviour specification associated with a stream interface template are either stream actions or internal actions. As above, actions in the behaviour specification relate states found in the history of states associated with that interface.

Modelling Signal Interface Templates in Z

Signal interface templates may be represented by:

SigIntTemplate

$$
\begin{array}{l}
signals : SigIntSig \\
sigbs : behspec \\
sighis : history \\
\hline
\forall s_1; s_2 : State;\ ss : \mathrm{seq}\ State \mid ss\ \neg\langle s_1; s_2\rangle \in \mathrm{prefix}\ sighis \bullet \\
\quad (\mathbf{let}\ sigActs == \{sig : SignalSig \mid sig \in signals.sigs\} \\
\qquad isSigAct(sig; s_1; s_2) \in \mathrm{dom}\ sigbs \cup \mathrm{ran}\ sigbs \bullet isSigAct(sig; s_1; s_2)\} \bullet \\
\quad (\mathbf{let}\ otherActs == \{ia : Internal \mid isIntAct(ia; s_1; s_2) \in \mathrm{dom}\ sigbs \cup \mathrm{ran}\ sigbs \bullet \\
\qquad\quad isIntAct(ia; s_1; s_2)\} \bullet \\
\qquad \langle sigActs; otherActs\rangle \in \mathrm{partition}\ \mathrm{dom}\ sigbs \cup \mathrm{ran}\ sigbs))
\end{array}
$$

This states that the only actions that can be found in the behaviour specification associated with a signal interface template are either signal actions or internal actions, and that these actions relate states in the history of states associated with that interface.

The specification fragments developed so far have dealt with the syntactic and behavioural aspects of computational interfaces in isolation. The computational viewpoint also prescribes rules as to how these interfaces can be composed with one another and also substituted for one another. These rules and how they might be formalised are considered in more detail in the following chapters.

5.2 Formalising Other Viewpoint Languages

The above sections have shown in detail how LOTOS and Z can be used to formalise the computational viewpoint language. This is only one of the ODP viewpoints. We consider now briefly how LOTOS and Z might be used, or not as the case may be, to formalise the other viewpoint languages.

5.2.1 Issues in Formalising the Enterprise Viewpoint Language

The enterprise viewpoint language comprises those concepts and structuring rules that are necessary for describing the purpose, scope and policies that a given enterprise should abide by. Surprisingly, the enterprise viewpoint language is very short with few concepts and structuring rules that apply to them. At the time of writing the development of a more complete enterprise language is undergoing standardisation [114].

>From a formal viewpoint, it is obvious that not all statements that might be made in the enterprise viewpoint can be formalised directly. Indeed, scope and purpose are inherently informal. The statement, “ this system implements the traffic signalling for railway X”, is a typical example of a statement dealing with scope and purpose. Whilst not directly usable for specifiers, such statements are nevertheless important to document. Normally, this is done in the informal commentary that accompanies every specification.

It is not possible to formalise anything without first having statements about the system under consideration. Statements are often made in natural language and as such are open to interpretation and possibly ambiguity. Formal methods offer precision and clarity in defining statements. These statements often take the form of business rules, captured in ODP by the notion of *policy*. Policies contain sets of rules related to a specific purpose. These rules may be:

- **obligations:** prescriptions that certain behaviours are required;
- **permissions:** prescriptions that certain behaviours are allowed to occur;
- **prohibitions:** prescriptions that certain behaviours must not occur.

Permissions and prohibitions may be modelled directly in LOTOS and Z. In LOTOS, for example, a permission might be represented simply through the temporal ordering of actions, *e.g.* an action that must occur before some other behaviour can occur. Alternatively, selection predicates and guards can be used to model permissions. Modelling prohibitions in LOTOS can be achieved in several ways also. For example, if a given behaviour is not permitted to occur then it is possible in LOTOS to simply not specify that behaviour, *i.e.* ignore prohibited behaviours. Alternatively, if it is likely that a given prohibited behaviour can have at some later time a permission associated with it, then a prohibition may be modelled through a behaviour expression following an identified permission event offer. That is, the behaviour is prohibited until the identified permission event offer occurs. Alternatively, a guard or selection predicate may be used to model a prohibition.

In Z, all behaviours are modelled as permissions. That is, possible behaviours are specified with operation schemas which may or may not occur depending upon any preconditions and postconditions that might be associated with given operation schemas and the environment of the specification. Prohibited behaviours in Z correspond to behaviours which would leave the specification in an indeterminate state, *e.g.* through violating an invariant. As such, prohibited behaviours are generally avoided in Z through providing total operations.

Neither language is well suited to modelling obligations though. Both LOTOS and Z allow behaviours to be specified but it is not generally the case that prescriptions on the occurrence of these behaviours is given. This may sometimes be done informally. For example, consider the LOTOS behaviour expression **a; b; stop**. It can be said that event (behaviour) **a;** must (is obliged) to occur before event **b;** can occur. However, generally speaking nothing obligatory can be said about event **a;**. If the only possible event in the whole specification is **a;**, however, then it can be said that **a;** will eventually occur. Making statements about obligations in LOTOS (and Z) is thus not generally possible as there is usually more than one action which can occur at any given time.

>From this discussion, it can be seen that LOTOS and Z have advantages and disadvantages in formalising enterprise rules. The problem remains though with regard to our interest in developing an architectural semantics that the exact nature of the rule is not given. As an example, consider the rule: the number of users logged on is less than ten. Both LOTOS and Z can formalise this rule, with Z more likely to be less verbose, *e.g.* $Users \leq 10$. Such requirement statements can be captured, but to be formalised they must exist in the first place. As a result, developing an architectural semantics for the enterprise viewpoint language is limited due to the lack of prescriptivity.

Similar problems are found with the other concepts in the enterprise viewpoint language. For example, a domain is defined as a set of objects related by a characterising relationship to a controlling object. In LOTOS, this may be achieved through a style of specification which captures a given characterising relationship between a set of controlled objects and a controlling object. In effect this will mean that the objects being controlled have parts of their behaviour expressions in common with one another, *i.e.* they share some similarities in their behaviour expressions and these similarities define the relationship through which they are controlled. It is likely that a constraint-oriented style of specification is best suited for capturing this configuration. Consider the following LOTOS fragment showing how a simplistic security domain might be achieved within LOTOS.

```
     Controller[g1](idset, Sec_Level) |[g1]| (Obj[g1](id1,1) ||| Obj[g1](id2,2))
where
     process Controller[g1](idset:IDSET, Sec_Level:Nat)
```

```
:noexit:=
  g1 ?sl: Nat ?id :ID[id in idset];
  ([sl ge Sec_Level] -> g1 !ok !id; Controller[g1](idset,Sec_Level)
   []
   [sl lt Sec_Level] -> g1 !error !id; Controller[g1](idset,Sec_Level))
endproc (* Controller *)
process Obj[g1](id: ID, Lev:Nat)
:noexit:=
  g1 !Lev !id; g1 ?res: Result !id;
  ([res eq ok ]-> ...(* behaviour *)
   []
   [res eq nok]-> ...(* other behaviour *))
endproc (* Obj *)
```

In this example, all of the objects have a similar relationship with the controlling object which knows their identities. This object checks whether any object it controls has enough authority to perform a certain operation which has not been further specified. As may be seen, the modelling of domains in LOTOS can be achieved provided a style of specification is followed. However, it is not the case that domains are an inherent feature of the LOTOS language.

Z may also be used to specify domains to a certain extent[5] but, as with LOTOS, specific examples of domains are required, *i.e.* explicit characterising relationships should be provided.

>From these discussions, the development of an architectural semantics for the enterprise viewpoint language is limited. LOTOS and Z require specific rules and instances of concepts to be given. Unfortunately, such prescription in ODP is not[6] given. As a result, the development of an architectural semantics for the enterprise viewpoint language is reduced to informal modelling suggestions.

5.2.2 Issues in Formalising the Information Viewpoint Language

ODP defines information as:

Any kind of knowledge about things, facts, concepts and so on, in a universe of discourse that is exchangeable amongst users.

Although information will necessarily have a representation form to make it more communicable, it is the interpretation of this representation (the meaning) that is relevant in the first place.

To give a representation-free way of modelling information, [109] uses the notion of *invariant, dynamic* and *static schema*. These represent the invariant properties of information, the dynamic properties of information, *e.g.* the legal manipulations of data, and the state and structure of information at some particular time respectively.

We consider how these concepts may be represented in LOTOS and Z.

Formalising the Information Viewpoint Language in LOTOS

Generally, in LOTOS the information to be modelled in a specification is represented in Act One. This is manipulated and used in the process algebra part. Information might also be modelled using the process algebra part of LOTOS. However, the above definition of information in ODP states that the information must be exchangeable amongst users. In LOTOS, exchange of data can only occur between instantiations of process definitions using data modelled in Act One. It is not possible in LOTOS to exchange process definitions between process definitions. As a result a given information item is represented by an instance of an Act One sort with associated operations and equations. Instances of sorts in the process algebra are typically simply values. To model information items that have a form of existence requires process algebra specification styles to be adopted. Examples of these include recursion in *process definition*s,

[5] In fact Z is more suited to modelling domains since it allows global predicates to be written that should be satisfied by all operations in the specification. The domain here does not explicitly model a controlling object as such, rather it models the effects of the rules imposed by the controlling object.

[6] and should not be, if a multitude of systems based on ODP are to be constructed.

where the information item is an element of the *value parameter list* associated with that *process definition*. Alternatively, **let...in** clauses can be used to model information items with a form of existence.

>From this it can be seen that the whole Act One part of a LOTOS specification represents an ODP *invariant schema* that instances of sorts and their manipulations declared in the process algebra part of LOTOS must satisfy. This is slightly strange (but correct!) in that the equations and operations in Act One are not "watchdogs" looking over illegal information manipulations; they define the manipulations and so cannot be wrong in themselves!

An ODP *static schema* is represented by the binding (value) associated with the name of an instance of an Act One sort at some point in time, *e.g.* an Act One sort *Queue* might be initially bound to the value *empty*.

An ODP *dynamic schema* is given by the Act One expressions used to modify information objects (Act One sorts) in the process algebra part of a LOTOS specification. It should be pointed out that it is Act One expressions that are associated with the action denotations of the behaviour expressions that manipulate the information. For example, consider the event offer *g !push(val,empty_queue)* that might exist as part of the behaviour of a given specification. It is not the event offer itself that manipulates the information item, in this case an empty queue, but the Act One expression *push(val,empty_queue)* that manipulates the information. Thus it is only Act One that manipulates the information; however, the Act One can only be used in LOTOS when it is associated with behaviour, *i.e.* the process algebra. Thus the notion of information processing activity in LOTOS is linked both to the Act One information modelling, and to the Act One information processing contained in the process algebra part of LOTOS.

One way of considering this is that the process algebra represents a behavioural framework on which information can be hung and manipulated.

Formalising the Information Viewpoint Language in Z

Information is given in Z through mathematical data types. The most elementary of information items in Z are the basic types. From these, more complex (composite) types may be developed. There are three main composite types in Z: set types, Cartesian product types and schema types. These three types and the basic types may be combined and manipulated using type constructors which follow mathematical rules to model the different information possible in a Z specification. Examples of these type constructors include functions, relations, etc.

The application of the four types mentioned and any new types developed using mathematical type constructors categorise the information items into sets. The type (*ODP notion*) as discussed in section 4.3.7 of any given information item is then determined by set membership.

An ODP *invariant schema* can be written explicitly in Z. It corresponds to any invariants (predicates) that may be present in the state schema or axiomatic descriptions associated with a Z given specification. These predicates are independent of behaviour, or rather they govern the possible behaviours of the given Z specification, where behaviour in Z is modelled as the performance of operations defined in operation schemas.

An ODP *static schema* is represented in Z by the binding associated with the variables declared within the state schema of the specification at any point in time. Here time is an abstract notion which is related to the state of the specification. Time is thus dependent upon the state changes that have occurred and those that can occur within the specification.

An ODP *dynamic schema* is represented in Z by the possible behaviours that might be exhibited by the specification. A dynamic schema is given by the operation schemas present in a Z specification. All operation schemas must satisfy any global predicates (invariants) that may limit the bindings that can be taken by variables declared in the state schema or in axiomatic descriptions associated with the specification.

>From these discussions, it is obvious that LOTOS and Z are limited in what they can express from the information viewpoint. Both languages can model information. Z in particular is well matched to the ODP concepts. As with the enterprise viewpoint though, without specific statements, *e.g.* regarding the structure and legal manipulations of information, the development of an architectural semantics for this viewpoint is reduced to informal modelling suggestions. These issues were discussed also presented in [176].

5.2.3 Issues in Formalising the Engineering Viewpoint Language

The computational viewpoint allows models of systems as collections of interacting objects to be built. This offers a useful and powerful conceptual model for abstracting over the lower-level system considerations. Ultimately these lower-level considerations have to be addressed. The engineering viewpoint language provides the concepts and structuring rules for dealing with such issues.

Certain work, [68] and [203], has shown how LOTOS can be used to a certain extent to model aspects of the engineering viewpoint language. Certainly, a structuring of concepts can be achieved that reflects the text contained in the ODP-RM. Given that the engineering viewpoint is used to realise (to a certain extent) the models of systems developed in other viewpoints, we would argue that it is outside the scope of this thesis. Here we are more interested in the specification of distributed systems as opposed to the mechanisms used for realising the specifications developed.

5.2.4 Issues in Formalising the Technology Viewpoint Language

The technology viewpoint focuses on the requirements of the system with regard to specific hardware and software technologies that are to be used. It is possible to provide a form of formalisation here, *e.g.* a given application might make requirements regarding the hardware or software technologies that might be used. This information could be formalised as a tuple for example, *e.g.* $\langle HardwareX; OpSystemY; CompilerZ \rangle$. The benefits to be gained from such formalisation are questionable however.

5.3 Summary

This chapter has focused in detail on how the computational viewpoint of ODP can be formalised in LOTOS and Z. Specification templates have been generated where possible and a discussion has been given of the advantages and disadvantages of the two languages for specifying computational objects generally. We have also highlighted the issues involved in developing an architectural semantics for the other viewpoint languages. In the case of the enterprise and information viewpoints the main problem is a lack of prescriptivity in the concepts. These issues were discussed in more detail in [176] and [175]. The engineering and technology viewpoints we regard as being outside the scope of specification generally. Other works have attempted to focus in more detail on checking consistency between viewpoint languages, *e.g.* [31, 32, 183]. In the main though, these works are more involved with composition of specifications more generally, and not viewpoint language specifications. It is often the case that enterprise and information language specifications use informal or semi-formal notations, *e.g.* diagrams and text. As such checking and proving consistency between viewpoint specifications is unlikely to be possible in many situations.

It could be argued that to a certain extent we have used both LOTOS and Z in a contrived manner to specify computational interfaces. With LOTOS for example, we modelled interface references in Act One and interfaces themselves in the process algebra. To some extent this places a high level of requirements on the users of the architectural semantics. For example, they must ensure that the interface references are fulfilled by the processes they represent. With Z, we have an approach that does not correspond to Z in its classic style of state and operations. These models have not been selected without reason however. We shall see in chapters 7 and 8 that these models offer us much more flexibility to specify open distributed systems.

The issue of specification style reflecting the architecture under consideration versus good specification style is addressed in more detail in chapter 9. We note here that there is a definite tension between “good practice” and “working practice”. That is, concepts that may not be interpreted when using a language in its normal (classical) usage compared to concepts that can only be specified when using the language in a particularly stylised manner. The area of typing, subtyping and type checking are ODP concepts that, when formalised, reflect this conflict directly.

As discussed, computational interfaces consist of interface signatures, behaviour specifications and environment contracts. Computational objects consist of one or more computational interfaces, a behaviour specification that relates the behaviours in the interfaces, and an environment contract for the object as a

whole. In this chapter, environment contracts have largely been ignored. We discuss how they might be represented in the next chapter.

Chapter 6

Non-Functional Aspects of Distributed Systems

In this chapter we investigate how LOTOS and Z might be used to specify a collection of specification fragments dealing with non-functional aspects of systems. There are numerous non-functional aspects of systems that we might be interested in. It is not possible to formalise all of them. Instead, we show how three disparate non-functional aspects of systems might be formalised. Specifically, we show how LOTOS and Z can be used to specify aspects related to cost, resource usage and timing considerations.

We note here that these non-functional aspects are very likely to be modelled in other viewpoint languages. In particular, the enterprise and information viewpoint languages are very likely to model issues related to non-functional aspects of systems. As noted in section 5.2, the enterprise and information viewpoint languages do not provide enough prescription to develop specification architectures directly. As a result, we focus here primarily on how non-functional aspects can be represented directly in the computational viewpoint language. On the one hand this allows a greatly simplified approach to be taken since we do not deal with issues of consistency between viewpoint languages, [31, 22, 32], *e.g.* ensuring that the separate viewpoint descriptions are: non-contradictory, behaviourally compatible or in some sense conformant depending on the interpretation of consistency taken [31]. On the other hand it reduces the formal description of non-functional aspects of systems to possibly over-simplistic models.

We shall see in the following chapters how these specification fragments can then be used to influence type checking as might occur when composing systems or replacing one system for another. More generally, this will apply to writing specifications of distributed systems in LOTOS and Z.

6.1 Classifying Non-Functional Aspects of Systems

As discussed in section 1.2, consideration of non-functional aspects of systems is crucial if systems as a whole are to interwork correctly. We illustrate this through the following three examples which form the basis for discussions and formalisations in the rest of this chapter.

- If a system produces correct results outside a given deadline then it might be the case that this is as bad as producing incorrect results within a given deadline.
- Knowledge that a server is currently processing as many requests as it can may be used by a client to avoid sending yet another message (request) to the server. Similarly, such information can be used by a client to select a server that is less busy than other servers.
- Knowledge that a cost is attached to a server interface will undoubtedly be an influential factor in determining whether a client interacts with that interface. Similarly, the cost attached to interfaces can be used by clients to select a particular interface, *e.g.* the cheapest, from a set of available ones.

>From these examples, it is obvious that non-functional aspects are especially pertinent to distributed systems where the system as a whole is dependent upon the successful interworking of the interacting objects from which it is composed.

Despite this, non-functional aspects have been until recently a largely overlooked area of computing science. Certainly, the formalisation of a multitude of non-functional aspects of systems has not been a well investigated area. Whilst temporal aspects of systems have been formally investigated to a certain extent, *e.g.* timed extensions to LOTOS [153] and temporal logics generally [135, 160], other factors that can influence behaviour, *e.g.* cost, have not been considered in any great depth. For example, consider the cost of using an interface or an operation that exists in an interface, or the maximum number of invocations that an interface can deal with at any one time. This information, whilst not directly representing the behaviour of that interface, is nevertheless vitally important if another object wishes to access that interface and have some form of desirable behaviour.

There are several immediate questions that arise when specification of non-functional aspects of systems is considered. These are:

- what are they?
- how are they represented?
- how are they checked?

With regard to the first bullet point, as stated previously, we focus here on cost, time and resource based non-functional aspects. Other aspects, *e.g.* reliability and fault tolerance, could equally well be specified though. The next bullet point has numerous answers which reflect directly on the third bullet point. We identify three different ways of representing non-functional aspects of systems, each of which has a valid role in relating non-functional aspects to behaviours and interfaces more generally.

- a non-functional aspect is extra information that does not relate directly to the behaviour with which it is associated, but may influence that behaviour indirectly. We shall see that cost is an example of such extra information.
- a non-functional aspect is an assertion derived from the behaviour with which it is associated.
- a non-functional aspect is an assertion that limits the behaviour with which it is associated directly.

These last two bullet points are closely related. In the second point, the assertion requires a detailed knowledge of the behaviour which then allows predicates on this behaviour. One example of this type of non-functional requirement is that the maximum delay associated with the production of an information flow is less than some value. With the third bullet point, the non-functional aspect is more of a predicate that dictates the behaviour, rather than the behaviour dictating the predicate. One example of this kind of non-functional aspect representation is the maximum number of invocations an interface can process at any one time. We shall see how LOTOS and Z can be used to model these different interpretations.

The next question is where do non-functional aspects exist. They might for example, be found in all operations that comprise the interface on which the assertion is made, or elsewhere such as some management interface.

It should also be possible to modify or withdraw assertions made on behaviours. The modifications themselves may take place when (functional) behaviour occurs, *e.g.* when invocations are received, or only through specific management operations say, *e.g.* to increase the cost of accessing an interface.

Non-functional aspects associated with an interface may or may not be checked, although cost is very likely to be one area where checks are made by both the interface provider and the accessor of the interface. Checks may be performed directly each time an operation is to be invoked, *e.g.* through inquiring about the current values of the non-functional aspects under consideration. In this case the satisfaction of a check might mean an answer whose value is within the expected limits for that non-functional aspect. Alternatively, checking can be done only once when interfaces are *bound* and all subsequent interactions at the interface are based on that initial check. We shall investigate this issue in more detail in the following two chapters.

6.2 Specifying Selected Non-Functional Aspects in LOTOS

In the previous chapter we showed how an interface reference might be represented in LOTOS by an Act One data type whose structure was given by:

```
type IRef is Location, OpSet, Constraints
     sorts IRef
     opns ...
          getCons: IRef -> Constraints
endtype (* IRef *)
```

This data model was then used to restrict the interactions that occurred between processes. The restrictions we identified in the previous chapter were based primarily on the set of operations associated with the interface reference. In this chapter, we show how the *Constraints* data type can be represented and, as we shall see in the next chapter, how it can be used to influence (restrict) the interactions that can take place between computational objects. We consider firstly cost-based non-functional aspects.

6.2.1 Cost-Related Non-Functional Aspects in LOTOS

Costing generally is concerned with charging for the use of, and access to, services. Costing for the use of services in a distributed system can be a complex activity. For example, issues related to security, *e.g.* to check for authenticity of users, and different charging policies associated with different domains might have to be dealt with. A cost in general is a monetary value which we represent here as a constructor operation that returns a cost constraint[1] along with an operation (*Ord*) that takes a cost constraint and returns a natural number. We also introduce a comparison operator (*cheaper*) between cost constraints. This checks whether the value of one cost constraint is less than another. We also introduce two operations to increase and descrease costs.

```
type CostConstraint is NaturalNumber, Boolean
     sorts CostCons
     opns cost            : -> CostCons
          _cheaper_       : CostCons, CostCons -> Bool
          incC,decC       : CostCons, Nat -> CostCons
          Ord             : CostCons -> Nat
     eqns forall c1, c2: CostCons
          ofsort Bool
             c1 cheaper c2 = Ord(c1) le Ord(c2);
endtype (* CostConstraint *)
```

For brevity we omit the equations for increasing and decreasing costs. It is quite possible to model costs directly as natural numbers but we leave them represented as constraints for clarity. We shall see that it is possible to use these cost constraints when deciding whether two interfaces can be bound to one another or replacing one interface for another. The issues related to the charging itself are not considered. For example, it might be the case that access to an interface results in a set charge established once when the interface is accessed. It might also be the case that the costs incurred by accessing an interface are cumulative and are based on how long the interface is accessed. Combinations of these possibilities are also conceivable, *i.e.* an initial access charge is made along with a cumulative charge based on how long the interface is accessed.

A simplistic charging policy based on access to an interface only may be represented by the following LOTOS fragment:

```
process Object[ g...](myRef: IRef,...costInterface: CostCons,...):noexit:=
 g ?opName: Name !myRef ?pl: HOPList;
 ([ getCC(getCons(getIRef(pl))) cheaper costInterface ] ->      ...(* error message *)
  []
  [ not(getCC(getCons(getIRef(pl))) cheaper costInterface) ] -> ...(* other behaviour *))
  []
  g ...!costInterface...; Object[ g...](myRef,...costInterface,...)
endproc (* Object *)
```

[1] In effect the constraint arises when comparison of cost values is given, *e.g.* checking whether one cost value is greater than another.

We shall discuss the operation *getCC* later in this chapter. The operations *getCons* and *getIRef* are discussed in more detail in the following chapter. We note here that we no longer have a simple parameter list being sent. This parameter list is higher order in that it can contain references to other interfaces. We discuss this higher order type system in more detail in section 7.4.1. It suffices to say here that it is possible to take a parameter list and return the constraints associated with it; the constraint here is cost. This specification fragment states that if the cost constraint associated with the parameters of the invocation is less than the value charge for that interface, then access is prohibited to the interface behaviour.

It is quite possible to model more complex cost constraints. For example, processes may have a variable used to model the money available and requests sent may decrement this variable say. Subsequent requests may only be sent depending on the current value of this variable. With this model it is possible to state constraints such as the money in the system remains constant. The actual representation of money and the charging process itself is of course only an abstraction of the real world process of charging.

It might be the case that sets of cost constraints are applied to a particular interface. This is likely to be the case if individual operations have costs associated with them. Sets of costs constraints might be represented by:

```
type CostConsSet is Set actualizedby CostConstraint
     using sortnames CostConsSet for Set CostCons for Element Bool for Fbool
endtype (* CostConsSet *)
```

Other models of collections of costs constraints that might be associated with the operations in an interface are possible. For example, sequences of cost constraints might be given and operations in an interface assigned cost constraints based on the positioning of cost constraints in this sequence.

6.2.2 Resource-Related Non-Functional Aspects in LOTOS

A resource may be regarded as something which can be accessed and used by objects in a distributed system. Interfaces and the services that they are used to model may be regarded as resources which can be accessed by other interfaces in the system. Where this is the case, it is often useful to be able to determine the usage of interfaces. Interfaces and the computing systems on which they exist only have a limited resource capacity, *e.g.* processing power and available memory. As such, the number of requests that an interface can satisfy at any given time should be limited. Objects wishing to interact with that interface may well wish to know what the upper limit is on the number of requests that can be processed and also how many requests are currently being processed. This information may then be used to determine whether one particular interface over another is selected. Further, for systems that have some form of resource allocation policy, the number of requests being processed by an interface at a given time may determine which interface receives a request sent to that system.

It is possible to model resources at many levels of abstraction. For simplicity, we model resources simply as constraints with an operation (*morethan*) to check whether one resource constraint has more resources than another.

```
type ResourceConstraint is NaturalNumber, Boolean
     sorts ResCons
     opns res             : -> ResCons
          _morethan_      : ResCons, ResCons -> Bool
          incR,decR       : ResCons, Nat -> ResCons
          Ord             : ResCons -> Nat
     eqns forall r1, r2: ResCons
          ofsort Bool
             r1 morethan r2 = Ord(r1) gt Ord(r2);
endtype (* ResourceConstraint *)
```

As with cost constraints we introduce two operations for increasing and decreasing the resources available. For brevity, the equations are omitted. We show now how a resource constraint might be used to limit the behaviour associated with an object. For simplicity, we focus on an operational interface composed of interrogations only.

```
process Object[ g...](...currentRvalue, upperRlimit: ResCons, ...):noexit:=
        hide ac in
         AccessControl[ ac ](currentRvalue, upperRlimit)
         |[ ac ]|
         FunctionalBehaviour[ g, ac, ...]
  where
    process AccessControl[ ac ](currentRvalue, upperRlimit:ResCons):noexit:=
        ac ?opName: Name;
        ([ (opName eq invName) and (currentRvalue lt upperRlimit) ] ->
                ac !ok; AccessControl[ ac ](incR(currentRvalue,1),upperRlimit)
         []
         [ (opName eq invName) and (currentRvalue ge upperRlimit) ] ->
                ac !nok; AccessControl[ ac ](currentRvalue,upperRlimit)
         []
         [ opName eq termName ]->
                AccessControl[ ac ](decR(currentRvalue,1),upperRlimit))
    endproc (* AccessControl *)

    process FunctionalBehaviour[ g, ac ...]:noexit:=
       g ?invName: Name !myRef ?pl: PList;
       ac !invName;
       ac ?res: Message;
       ([ res eq nok ] -> g !termName !SomeRef !nok;
                          FunctionalBehaviour[ g, ac ...]
        []
        [ res eq ok ] -> (  (* process request *)
                            ac !termName;
                            g !termName !SomeRef !results;
                            stop
                          |||
                            FunctionalBehaviour[ g, ac ...]))
    endproc (* FunctionalBehaviour *)
endproc (* Object *)
```

Here an object template fragment is given that has two resource constraint values associated with it. The first of these represents the current number of invocations being processed and the second represents the maximum number of invocations that can be processed at one time. Invocations are only accepted if the current resource value is less than the upper limit. This limit and the modifications to it are kept in a process dealing with access control. If this returns an affirmative response to an invocation then a behaviour is spawned to deal with the request and at the same time a recursive call made to accept more invocations. We note here that we state that following the termination the spawned process stops. This may be changed if for example the spawned process modifies some state (represented as Act One sorts in the formal parameter list). If a negative response is returned from the access control process then an appropriate error message is issued in the associated termination.

It should be pointed out that this model of resource counting treats all actions as being equivalent resource consumers. This may not always be a valid assumption. It might be the case that certain invocations result in a greater increase in the resource count than others. This depends to a great extent on the specification itself and what the actions in the specification are modelling. For example, issues related to concurrent access to shared memory where locks might be set will definitely have an effect on the simplistic model of resources based solely on action counting.

The above specification fragment also does not deal with objects that interact with other objects to satisfy the requests that have been received. This might cause the resource count to be reduced for example.

It is likely that features for determining the maximum limit and current value of resource usage will be available. It might also be the case that relationships between these values might be given, *e.g.* to deal with fair allocation of resources say by a resource allocation manager.

It might be the case that we have sets of resource usage constraints associated with an interface. These might be represented by:

```
type ResConsSet is Set actualizedby ResourceConstraint
     using sortnames ResConsSet for Set ResCons for Element Bool for Fbool
endtype (* ResConsSet *)
```

It is also possible to assign resource constraints to each operation in an interface directly. This might be achieved through modelling the resources associated with an interface as a sequence of resources and assigning resources to operations based on some positioning in this sequence. More complex resource assignment policies are also possible, *e.g.* where operations share resources.

6.2.3 Time-Related Non-Functional Aspects in LOTOS

LOTOS was developed specifically for dealing with the temporal ordering of events. As such, it does not attempt to deal with quantitative timing considerations where events are expected to occur at certain specific times or within specific intervals. Nevertheless, we have seen in section 5.1.2 how LOTOS can be used to model information flows where timing considerations are especially important. This was achieved through modelling the sequence of time ordered information items in Act One and establishing their time by the occurrence of events in the process algebra. Whilst allowing timing issues to be reasoned about we acknowledge that we only have an abstract model of time, *i.e.* the time stamps we associate with events are only an abstraction of the real world location in time at which an event can occur. Thus if an event is time stamped to occur at time *t* say, then there is no inherent feature of LOTOS to ensure that this event is fired to occur at this time.

As discussed, one issue with this model is that time may not be global to all processes. This need not necessarily be a bad thing. For example, the time at which separate systems process requests may not always be global. Given the limitations of measuring the absolute value of real time[2] computers will undoubtedly have slightly different time clocks. Synchronisation of these clocks may thus be necessary.

This model requires that processes with time dependencies have a variable representing the current local time. This time variable may be used to set the occurrence times of actions as well as restricting the occurrence actions to certain times or time windows. We introduce now the set of time constraints that will be used in establishing time relationships between actions.

```
type TimeConstraints is NaturalNumber, Boolean
     sorts TimeCons
     opns makeT,maxT,minT : -> TimeCons
          _+_,_-_         : TimeCons, Nat -> TimeCons
          Diff            : TimeCons, TimeCons -> Nat
          Ord             : TimeCons -> Nat
          _eq_,_gt_,_lt_,
          _le_,_ge_       : TimeCons, TimeCons -> Bool
     eqns forall t1,t2, t3: TimeCons
          ofsort Nat
             (t1 eq t2) => Diff(t1,t2) = 0;
             (t1 gt t2) => Diff(t1,t2) = succ(Diff(t1,succ(t2)));
             (t1 lt t2) => Diff(t1,t2) = Diff(t2,t1);
endtype (* TimeConstraints *)
```

We omit many of the equations for brevity. It is likely also that sets of timing constraints will exist. These may be represented as:

```
type TimeConsSet is Set actualizedby TimeConstraints
     using sortnames TimeConsSet for Set TimeCons for Element Bool for FBool
endtype (* TimeConsSet *)
```

Duration of Actions in LOTOS

As discussed in section 4.2.1, actions in LOTOS are atomic. As such it is typically not the case that issues related to their duration are considered. Nevertheless, it is possible to specify the duration of a given action provided the specification style introduced in chapter 5 for modelling multimedia flows is used. For example, an action that has a duration of three time units might be represented by:

```
process TimedProcess[ g....](myRef: IRef, ... tnow: TimeCons...):noexit:=
     g ?op: Name !myRef ?pl: PList;
```

[2] *e.g.* to the accuracy given using a caesium clock say.

```
     (* checks on validity of parameters *)
     TimedProcess[ g....](myRef, ... (tnow+3)...)
endproc (* TimedProcess *)
```

Here the variable representing time *tnow* is increased by three time units. For simplicity we assume here and elsewhere that the natural numbers 1,2,3... etc are given, *i.e.* as opposed to representing 3 by succ(succ(succ(0))) say. Not all action have a duration though; actions may be atomic and hence instantaneous. An instantaneous action may be modelled simply by ensuring that the current time variable is unchanged. Thus the recursive call given here might be represented by:

```
TimedProcess[ g....](myRef, ... tnow ...)
```

It should be noted that this model of timed actions assumes that the actions themselves are simplistic event offers followed by a recursive call. This simple model may not always be applicable. In object-based LOTOS specifications [44], it is typically the case that internal events are used to process the requests from the environment. It is quite possible to specify systems so that timing aspects can be associated with any action, *i.e.* internal or observable. Alternatively, it is frequently the case that timing constraints on actions are restricted to observable actions only and assumptions such as maximum progress or action urgency (see section 3.3.3) are used to avoid dealing with timing issues of all actions. This is often justified by the assumption that internal actions have minimal times in comparison to the delays in waiting for observable actions to occur.

This model of action durations is straightforward when the actions themselves are composed through the choice operator. Some consideration of actions composed through the interleaving or parallel composition operator is required however. Consider two actions interleaved with one another that have action durations of one and three time units respectively. These might be represented by:

```
process TimedProcess[ g....](myRef: IRef, ... tnow: TimeCons...):noexit:=
     g ?op1: Name !myRef ?pl1: PList;
     (* checks on validity of parameters *)
     TimedProcess[ g....](myRef, ... tnow+1 ...)
     |||
     g ?op2: Name !myRef ?pl2: PList;
     (* checks on validity of parameters *)
     TimedProcess[ g....](myRef, ... tnow+3 ...)
endproc (* TimedProcess *)
```

Here the event offers may occur in either order. Unfortunately, the timing introduced for these actions means that they are no longer truly temporally independent. That is, it should be possible for both actions to occur within three time units if they were truly concurrent. Here the time for both to occur is four time units though. Another problem with this process is that there is no notion of a common time. That is, since the behaviours are interleaved with one another, once they occur they will never rejoin. This can be overcome through the use of the choice operator instead of interleaving, but this changes the behaviour of the process so that it blocks other behaviours once an event offer has occurred.

An alternative approach is to model the events with exit functionality.

```
process TimedProcess[ g....](myRef: IRef, ... tnow: TimeCons...):noexit:=
     (g ?op1: Name !myRef ?pl1: PList;
         (* checks on validity of parameters *)
         exit(tnow+1, any TimeCons)
     |||
      g ?op2: Name !myRef ?pl2: PList;
         (* checks on validity of parameters *)
         exit(any Time, tnow+3))
           >> accept t1,t2: TimeCons in (* choose larger time *)
              TimedProcess[ g....](myRef, ... tnow+(* larger time *)...)
endproc (* TimedProcess *)
```

Here the event offers may occur in either order. Unfortunately, it is not the case that two *op1* occurrences can occur within two time units say. That is, the behaviours must synchronise on **exit**. As a result, both operations must occur within three time units.

These problems arise from the limitations of modelling true concurrency in LOTOS. In LOTOS all events are interleaved with one another. As such, any model of duration in LOTOS is limited to an abstract representation. Despite this it is possible to specify and model certain timing aspects of systems and the constraints they impose on synchronisations. We consider now timing considerations related directly to information flows. These may be regarded as the most stringent of timed behaviours, *i.e.* they place most demands on the timing of events.

6.2.4 Multimedia Based Non-Functional Aspects in LOTOS

Timing of events is especially relevant to (distributed) multimedia systems when modelled in LOTOS. We have seen already how time stamps might be associated with information items in an information flow. It is often the case that restrictions are imposed on the times at which these information items are received. We investigate several types of restrictions and how they might be represented in LOTOS.

Maximum Jitter in LOTOS

Jitter may be regarded as the upper and lower limit on the time window at which a consumer can accept a frame. For example, if a frame is expected every t seconds with an allowed variation of Δt, then a frame should arrive within the range $t - \Delta t$ to $t + \Delta t$. There are two cases of jitter that we consider here: bounded jitter and unbounded jitter. The distinction between the two cases is dependent upon whether the arrival time of the last frame influences the arrival time of the next frame. In unbounded jitter, if frames are expected every t seconds with a variation of Δt then should frames consistently arrive early, but within the allowed time range, then the flows will eventually drift out of synchronisation. For example if t was 30 time units say and Δt was 5 time units and frames arrived every 29 time units, then after five frames had arrived, all subsequent frames would be outside the allowed range, *i.e.* the next frame would be expected at 180 but would arrive at 174 time units which would exceed the maximum unbounded jitter rate of 5.

In bounded jitter if a frame arrives early but within the allowed time variation, then the arrival time of the next frame is time t after that arrival time. Hence using the above numbers, if a frame arrives at time 29 then the next one would be expected at time 59 and not 60. From this, it can be seen that bounded jitter does not allow flows to drift out of synchronisation.

Unbounded jitter was represented in section 5.1.2. This corresponded to the variable *limit* given in the process definition representing the consume action. It should be noted that with this model of production and consumption of flows, consideration of the time taken for consuming and producing single frames is critical. If both producer and consumer produce and consume frames respectively at the same rate, then ignoring issues of latency and lost frames for now[3], there should never be any jitter. It might well be the case that producers and consumers operate at slightly different speeds however, *i.e.* the modifications to the time variable in the process instantiation are not necessarily equal for the producer and consumer. Bounded jitter models situations where the time difference between production and consumption of frames is slightly different but within a certain limit. This may be represented as:

```
process Consumer[ g....]( ...recFrames: FrameSeq, jitter, tnow: TimeCons, ...):noexit:=
 g ... ?inf: Frame ...;
 ([ Diff(getTime(inf),tnow) gt jitter ] ->
            ...(* error behaviour, e.g. drop frame *)
  Consumer[ g....](...recFrames,jitter, (tnow+t)...)
  []
  [ (Diff(getTime(inf),tnow) le jitter) and ((getTime(inf) - tnow) gt 0) ] ->
            ...(* successful behaviour, e.g. display frame *)
  Consumer [...g...](...addFrame(inf,recFrames),jitter,(tnow+t+Diff(getTime(inf)-tnow)),...)
  []
  [ (Diff(getTime(inf) - tnow) le jitter) and ((getTime(inf) - tnow) le 0) ] ->
            ...(* successful behaviour, e.g. display frame *)
  Consumer [...g...](...addFrame(inf,recFrames),jitter,(tnow+t-Diff(getTime(inf)-tnow)),...))
endproc (* Consumer *)
```

[3] These are considered in more detail in section 6.2.5 and 6.2.5 respectively.

Here we state that if the maximum bounded jitter is exceeded then the frame is dropped and the local time incremented by *t* time units. If the frame arrives within the timing restrictions imposed by the jitter then two conditions arise depending on whether the frame arrives slightly early or late. If the frame arrives earlier than expected then the time variable *tnow* is modified by adding on the time for consumption and subtracting the amount the frame was early by. For example, assuming frames are expected every 30 time units and the current time is 60 if the next frame is time stamped 59 then the time variable *tnow* would be set to (60+30-1= 89). As a result, the next frame is expected at time 89.

Alternatively should the frame arrive later than expected but within the allowed variation, *e.g.* at time 61 then the time variable is modified to (60+30+1= 91). Hence the next frame is expected at time 91.

Minimum Delay between Frames in LOTOS

The minimum delay between the frames a producer produces may be specified directly in LOTOS. This corresponds to the minimum difference between time stamps associated with two frames in the sequence of frames to be sent. If production of all frames is isochronous then the minimum delay is equal to the maximum delay and is a constant. Typically, information flows can have their rate of flows increased or decreased. The minimum delay between two frames is inversely proportional to the maximum throughput of the flow, *i.e.* the smaller the delay between frames, the higher the throughput. The minimum delay for a producer may be represented by:

```
process Producer[ g....]( ...toSend: FrameSeq, tnow, maxRate: TimeCons ...):noexit:=
   g ... !<SetTime(tnow+maxRate,head(toSend))> ... ;
   Producer[ g....]( ...tail(toSend),(tnow+maxRate),maxRate...)
endproc (* Producer *)
```

Here we assume the existence of some upper limit (*maxRate*) on the flow of frames. The minimum delay between frames for a consumer flow, *i.e.* the maximum rate at which a consumer can consume may be represented by:

```
process Consumer[ g....]( ...recFrames: FrameSeq, tnow, maxRate: TimeCons ...):noexit:=
   g ... ?inf: Frame ...;
   ((* reject frame if outside timing constraints *)
    []
    (* accept (display) frame and recurse *)
   Consumer[ g....]( ...AddFrame(inf,recFrames),(tnow+maxRate), maxRate ...))
endproc (* Consumer *)
```

Maximum Throughput of Flows in LOTOS

The maximum throughput that a producer can produce and a consumer can consume is useful information that might be needed when deciding whether two flows can be bound to one another. As stated, the throughput is inversely proportional to the delay. Throughput is typically measured over a period of time and recorded after the events have occurred. It is possible to specify the throughput at any given time through introducing variables to represent the number of frames (*fcount*) that have been sent or received and the time period (*tPeriod*) over which they have been sent or received. For a producer this might be represented as:

```
process Producer[g,m...](...toSend:FrameSeq,tnow,rate,fCount,tPeriod: TimeCons...):noexit:=
   g ... !<SetTime(tnow+rate,head(toSend))> ...;
   Producer[ g,m....]( ...tail(toSend),(tnow+rate),(fcount+1),(tPeriod+rate) ...)
   []
   m !(fcount div tPeriod); Producer[ g,m....]( ...toSend,tnow,rate,fCount,tPeriod...)
endproc (* Producer *)
```

Here we send frames every *rate* time units. The maximum throughput is given when *rate* is set to its maximum value, *e.g. maxRate* as given previously, over an unspecified time period. In this specification fragment we also count the number of frames sent and the time period over which they have been sent. This information is then made available through the event offer *m !(fcount div tPeriod);* which might exist

as part of some management interface. It should be noted that for simplicity we assume the existence of the division operator, that does not exist in the standard natural number type.

The maximum throughput of information flows for a consumer may be represented similarly. We omit the specification text for brevity.

6.2.5 Relation of Transport Media to Timing Issues in LOTOS

In these specification fragments for multimedia flows, we have largely avoided the issues involved in the communications medium that connects the producers and consumers of information flows. The communication medium is very likely to have a great influence on the issues involved in sending and receiving frames (and messages more generally). We investigate briefly how LOTOS might be used to specify two aspects of the communications medium and how they relate to the specification fragments given above. Specifically we consider the delay of messages and the loss of messages.

Specifying Latency in LOTOS

The delay of messages due to the communications medium that connects producers and consumers is typically termed latency. Unsurprisingly given the above discussions, latency is likely to have a definite influence on the timing of flows of information.

We discuss briefly two simplistic models of delay in LOTOS followed by a more complex (and realistic) model. In classical LOTOS, delays are represented by timeouts. The most simplistic model of this for a communication between a producer and consumer is:

```
process Comm[ p,g...]:noexit:=
   p ...?inf: Frame ....; i; g ...!inf...;
endproc (* Comm *)
```

Here gates *p* and *g* represent receiving outputs from producers and sending results to consumers respectively. This model of delay is inadequate for expressing the real-time constraints required by information flows since the specific timing requirements imposed by the producers and consumers are not dealt with, *i.e.* the time the producer stamps the frame is the same as the time the consumer receives it[4]. Also, if the information flows produced are sent to consumers receiving other flows that are to be synchronised with these flows, then timing considerations cannot be guaranteed. An alternative is to model the delay as a static feature of the communications medium itself. This might be represented as:

```
process Comm[ p,g...](delay: TimeCons):noexit:=
   p ...?inf: Frame ....; g ...!SetTime(getTime(inf)+delay),inf) ...;
endproc (* Comm *)
```

This approach allows delays to be included on the frames arriving and sent, but it does not allow any form of reasoning about the delays or variation in these delays. This might be satisfactory in a simplistic model of a system where issues related to how delays in the communications medium are established are not considered. A more realistic (and hence complex) model that allows delays to be calculated to a certain extent may be specified.

Latency and delays generally are influenced by several factors. These include:

- the maximum throughput that the connection between the producer and consumer can achieve;
- the current usage of the connection between the producer and consumer;
- the protocols used in transfering the information;
- the size of the information itself.

It is quite possible to formalise all of these abstractly in LOTOS to generate the amount of delay incurred through the connection. For example, a simplistic model of delay determined by considering only the ratio between available and used resources might be represented by:

[4] Assuming that the consumer only receives frames from one source.

```
estimatedTime: ResCons, ResCons -> Nat
```

The estimated time of delay would be increased if the difference between the resources used and resources available decreases. A more accurate model of determining delays might be based on stochastic models and queueing theories, *e.g.* Markov models, queueing networks [187, 191]. Using this simplistic model, the communication medium between a producer and consumer might be represented by:

```
process Comm[ p,g,m...](tnow: TimeCons,resUsed,resAvail: ResCons,fs: seq Frame):noexit:=
    [ resAvail morethan resUsed ] ->
    p ...?inf: Frame ....;
    ( Comm[ p,g,m...]((tnow+1),incR(resUsed,1),decR(resAvail,1),addFrame(inf,fs))  (*1*)
      []
      ( let next: Frame = highestPriority(fs,tnow) in
      g ...!SetTime(getTime(next)+tnow,next) ...;
      Comm[ p,g,m...]((tnow+1),decR(resUsed,1),incR(resAvail,1),remFrame(next,fs))))
    []
    m ...!estimatedTime(resUsed,resAvail) ...;
    Comm[ p,g...](tnow,resUsed,resAvail,fs)
endproc (* Comm *)
```

Here we state that frames are accepted by the communications medium if there are resources available. We include the gate *m* which can be used by some management interface say, to output the estimated time of frame delivery based on current resource usage. We also state that the estimated time between acceptance to delivery of a frame can be established based on the resources available and resources used. This inquiry takes no time to complete. This is a legitimate assumption if the frames being sent are particularly large. It might also be the case that differing sized frames take different amounts of time to be sent and arrive. This could also be specified here, *e.g.* through guards at the beginning of line (* 1 *). For example, an operation such as *getSize* might be available on a frame which returns a natural number. This number (or some multiple of this number) might then be used to determine how much the time was incremented by in the recursive call. Further, if frames are in formats that require specific protocols or compression techniques to be accepted then this too might influence the time for their acceptance and delivery. For simplicity here we assume that all frames are of comparable sizes and require comparable times to be accepted and sent.

If resources are available then the frame is accepted and added to a sequence of frames being transferred. The time is incremented; for simplicity sake we assume that this takes one time unit. The resources available and used are then decreased and increased by one respectively. We then specify that frames are delivered based on some priority. This priority can be established in numerous ways, *e.g.* based on the difference between the estimated time for delivery of the frames being transmitted and the current time. For brevity, we assume the existence of the operation *highestPriority*. Once a frame has been selected and sent to the consumer, the local time is incremented again and the frames used and available are decreased and increased respectively. This frame is then removed from the set of frames to be sent.

As can be seen it is possible in LOTOS to specify communication delays at many level of complexity. The descriptions so far have not dealt with the the possibility of errors occuring and the loss of frames.

Specifying an Imperfect Communication Medium in LOTOS

Errors such as the loss of messages between processes are typically represented by the internal event operator *i*. As previously argued, the internal event is limited in its ability to model timing considerations that might be associated with the delay of a frame. The internal event may be used to specify the loss of frames though without adversely affecting the modelling of information flows. A communication medium that allows for the potential loss of frames may be represented as:

```
process Comm[p,g...](tnow: TimeCons, fs: seq Frame):noexit:=
    p ...?inf: Frame ....;
    ( Comm[ p,g...]((tnow+1), addFrame(inf,fs))
      []
      i; Comm[ p,g...]((tnow+1), fs)
      []
      [ fs ne makeFrameSeq ] ->
      g ...!SetTime(getTime(next)+tnow,next)...;
```

```
                Comm[ p,g...]((tnow+1),remFrame(next,fs)))
            endproc (* Comm *)
```

For brevity we do not specify how frame *next* is obtained. This might be done using the above frame selection policy, *i.e.* with operation *highestPriority* say. Here frames can be delivered to consumers provided frames have been sent to the communications medium by producers. If a frame is sent by a producer then either it is added to the sequence of sent frames or an internal event occurs that causes the loss of that frame, *i.e.* the frame is not added to the sequence of frames to be delivered. As previously, when frames are delivered the number to be delivered is reduced by one.

6.2.6 Non-functional Aspects in LOTOS

>From the above specification fragments, a general model of the constraints that might be associated with interfaces focusing on timing, cost and resource constraints can be represented by:

```
type Constraints is CostConsSet, TimeConsSet, ResConsSet
     sorts Constraints
     opns makeCons    : CostConsSet, ResConsSet, TimeConsSet -> Constraints
          getCC       : Constraints -> CostConsSet
          getRC       : Constraints -> ResConsSet
          getTC       : Constraints -> TimeConsSet
          _Satisfies_ : Constraints, Constraints -> Bool
     eqns forall cc: CostConsSet, rc: ResConsSet, tc: TimeConsSet
          ofsort CostConsSet
            getCC(makeCons(cc,rc,tc)) = cc;
          ofsort ResConsSet
            getRC(makeCons(cc,rc,tc)) = rc;
          ofsort TimeConsSet
            getTC(makeCons(cc,rc,tc)) = tc;
endtype (* Constraints *)
```

Here we introduce operations to return the cost, time and resource constraints associated with a general constraint model. Further we provide an operation to check when one set of constraints is satisfied by another set of constraints. We shall investigate the equations associated with this operation in the next chapter when we consider how constraints can be used to establish compatibility between systems.

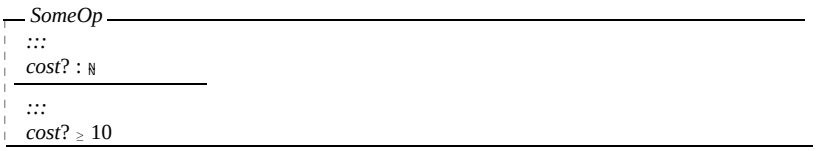
6.3 Specifying Selected Non-Functional Aspects in Z

In the previous chapter it was shown how computational interfaces could be represented by schemas consisting of syntactic aspects (signatures) that relate to behaviours and states. It was noted at the beginning of this chapter that consideration of syntactic and functional aspects of systems only may not be enough to ensure system interworking. As a result we extend the schemas representing interface templates in chapter 5 to include sets of assertions on the non-functional aspects of systems. In ODP termionology these collections of assertions are termed environment contracts. In this section we show how Z may be used to formalise the contents of these environment contracts. As with LOTOS we focus on cost, resource and timing considerations.

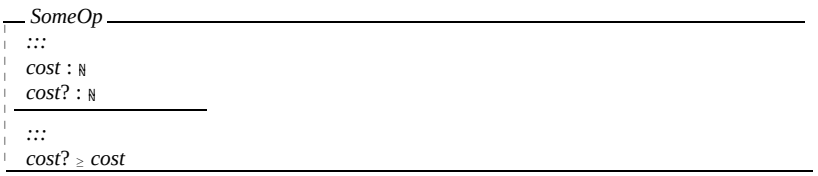
6.3.1 Specifying Cost-Related Non-Functional Aspects in Z

As discussed previously, cost is very likely to play an important role in checking that interfaces may interwork acceptably. We may separate two forms of costing: costing to use a single operation in an interface and costing to use the interface as a whole. This distinction implies that it is possible to have partial bindings, *e.g.* where a client only wishes access to a subset of the operations at a server. This selection might, for example be based on issues relating to the cost of using the operations in the interface as well as the functional behaviour those operations represent. We consider this issue in more detail in the following chapter.

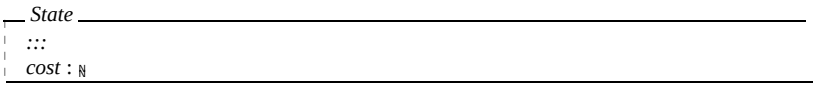
Cost constraints may be associated with operations or interfaces directly in Z. A cost might be represented simply by a variable, *e.g.* a natural number, with a predicate that this variable should be greater[5] than some value. This variable might be represented directly in an operation schema, *e.g.* as:



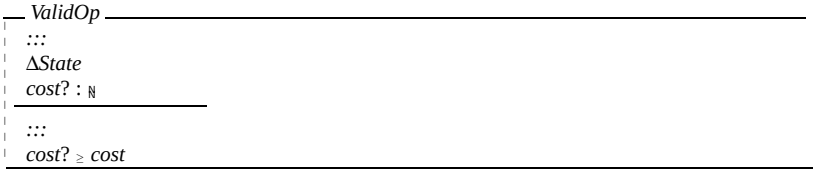
Here we state that the operation *SomeOp* has an invariant that requires the variable representing cost to be greater than 10. An improved model of costing would allow the costs to be varied directly, *e.g.* through modelling a variable for costs and having operation cost inputs to exceed this variable. If the cost is to be associated with the operation only and not applied to the whole interface, then this requires that the cost variable is declared locally to the schema representing the operation. This can be represented as:



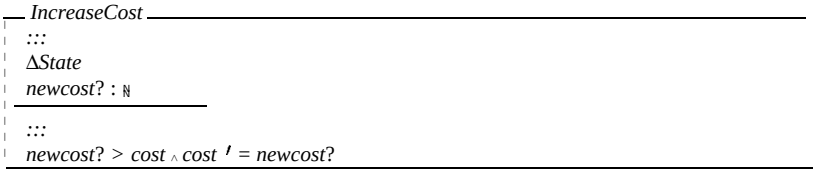
If the cost is to be associated with the interface as a whole then one way that this can be achieved is by including the cost variable in a state schema that is accessed by all operations comprising the interface. One such state schema might be represented as:



Costs of using the operations in the interface may then be modelled as:



Through this approach it is possible to model cost changes as:



Modelling upper and lower limits of costs is also trivial to achieve in Z. This can be done by simply modelling the cost as a range and requiring the cost inputs to operations to be within that range.

It is possible to have combinations of costs associated with operations and interfaces also. For example, a basic cost of accessing an interface and an additional cost of accessing certain operations. Using the above model of state this might be represented as:

[5] We consider the minimum cost of accessing an interface in the following discussions, but considerations of maximum costs require similar reasoning.

$$
\begin{array}{|l}
\textit{ValidOp} \\
\hline
:::\\
\Delta State \\
opcost : \mathbb{N} \\
cost? : \mathbb{N} \\
\hline
:::\\
cost? \geq opcost + cost \\
\hline
\end{array}
$$

The cost of accessing an operation should of course be less than the cost of accessing the interface as a whole, *i.e.* the cost of accessing all of the operations in the interface should be at least as great as the cost of accessing a single operation. This can be ensured through the use of predicates. If the cost of using an operation is increased then predicates should be given to ensure that the cost of using the interface is not exceeded. Alternatively, a new cost associated with the interface might be given.

It is also possible to model explicitly the cost constraints that might be associated with an operation or interface. This can be achieved through the use of functions or relations. If functions are used then this implies that interfaces (or operations) can only have one cost associated with them them at any one time. If relations are used then it is possible for interfaces (and operations) to have different costs associated with them at any one time. This might be an option if different classes of users exist with different privileges set for example. The choice depends on issues such as domain modelling. For an operational interface template with only one cost this can be represented as:

$$
\begin{array}{|l}
\textit{CostConsInterface} \\
\hline
setCostInt : OpIntTemplate \rightarrow \mathbb{N} \\
\hline
\end{array}
$$

It might also be the case that predicates are given here to ensure that this cost is greater than some value (or within some range of values). The cost of accessing (invoking) an interrogation signature in some operational interface might be represented as:

$$
\begin{array}{|l}
\textit{CostConsOperation} \\
\hline
\Xi CostConsInterface \\
setCostOp : OpIntTemplate \times IntSig \rightarrow \mathbb{N} \\
\hline
\forall oit : OpIntTemplate;\ is : IntSig \mid oit \in \mathrm{dom}\ setCostInt \wedge (oit; is) \in \mathrm{dom}\ setCostOp \bullet \\
\qquad setCostInt(oit) \geq setCostOp(oit; is) \\
\hline
\end{array}
$$

Here we state that the cost of using an interface should be greater than or equal to the cost of using an operation in that interface. Modelling changes in the cost of using interfaces or operations in those interfaces requires similar considerations as before, *e.g.* that the new operation cost does not exceed the interface cost say. An increase in cost of using an operation without increasing the cost of the interface may be modelled as:

$$
\begin{array}{|l}
\textit{IncreaseCostConsOperation} \\
\hline
\Delta CostConsOperation \\
oit? : OpIntTemplate \\
is? : IntSig \\
nc? : \mathbb{N} \\
\hline
setCostInt' = setCostInt \wedge (oit?; is?) \in \mathrm{dom}\ setCostOp \wedge \\
setCostOp(oit?; is?) \leq nc? \wedge nc? \leq setCostInt(oit) \wedge \\
setCostOp' = setCostOp \oplus \{(oit?; is?) \mapsto nc?\} \\
\hline
\end{array}
$$

Here we state that the new cost should be greater than the old cost and the new cost should not exceed the cost of using the interface. Alternatively the cost of using the interface might be increased also, in which case the predicates should be modified accordingly. We omit the Z texts for increasing the cost of an interface for brevity.

6.3.2 Specifying Resource-Related Non-Functional Aspects in Z

As with cost constraints, resource usage constraints may be modelled in Z through variables representing resources. To model resource usage in Z requires that actions that access and use resources have a variable

associated with them that is increased (or decreased) when the operation occurs. This implies that it is possible to distinguish between actions that use resources and actions more generally. This distinction may only be given with a higher level of prescription, *e.g.* with regard to how the specification is written and the models of resources generally. As with LOTOS, an abstract model of a resource might simply be represented by a natural number. The specification might then be written so that this number is limited by some amount. For example, if concurrent access to shared information is limited to some value, then actions that access this shared information can only occur provided this value is not reached. Consider the following model of information with access limited to some current number.

LimitedResource

$$resource : Resource$$
$$currentCount;\ maximumCount : \mathbb{N}$$

$$currentCount \leq maximumCount$$

Actions that access this resource are then required to ensure that the maximum level is not exceeded. The successful model of accessing such information might be represented by:

ValidOpAccessingResource

$$\Delta LimitedResource$$

$$currentCount < maximumCount \wedge currentCount' = currentCount + 1$$

Operations to increase or decrease the available resources may be specified directly through schema operations. We omit the Z text for brevity. It is possible to have other models of resource usage constraints in Z. For example, it might be the case that an upper limit is given on the number of invocations associated with interrogations[6] that can be processed at any one time say. Here we do not consider access to shared resources as such, rather we focus more on a model of action counting. This approach requires that schemas representing (interrogation) invocations have a variable to keep count of the number of invocations being processed. These schemas can only occur when this number is less than some upper limit. The occurrence of terminations will decrease this count by some value.

As with cost constraints it may well be the case that a more direct Z approach can be used to assign resource limits. This can be achieved through functions and relations[7]. Specifying that an operational interface template say has a resource limit can be specified by:

SetInterfaceResourceLimit

$$setIntLimit : OpIntTemplate \rightarrow \mathbb{N}$$

It might also be the case that operations could have resource limits set also, *e.g.* the maximum resources that this operation can get access to. This may be represented similarly. These models allow for direct assertions to be made on the interfaces (or operations). This is only really meaningful when the interfaces and operations themselves adhere to these limits. We shall discuss this feature of the Z language in section 6.4.

A third approach to modelling resource usage constraints can be achieved through the abstract Z approach given in chapter 4. For example, resource usage policies might state that once an invocation has been received, only a certain amount of action occurrences (or state changes) are permitted before an associated termination is to be sent. This information might be used to ensure that certain fairness criteria are upheld, *e.g.* through refusing more invocations if a termination has been blocked for a certain time.

To establish the maximum number of actions between an invocation and termination, we require two functions. The first of these allows us to determine whether an action is between two others in a behaviour specification. The second returns the set of actions between two actions in a behaviour specification.

Establishing whether an action is between two others in a behaviour specification can be represented as:

[6] We focus on interrogations here as invocations are required to produce terminations, *i.e.* more activity is required. The invocations associated with announcements may or may not require more activity and as such their occurrence does not necessarily increase the resource usage.

[7] It is less likely that interfaces will have several possible resource limits, but not impossible.

$$isBetween : (action \times action \times behspec) \leftrightarrow action$$

$$\forall a; b; c : action;\ bs : behspec \mid ((a; b; bs); c) \in isBetween \bullet (a; b) \in (\{b\} \lhd bs)^+ \land (c; b) \in (bs \rhd \{a\})^+ \Rightarrow a = c \lor (a; c) \in (bs \rhd \{a\})^+$$

This states that if *a* can cause *b*, *i.e.* (*a*,*b*) is in the transitive closure of the behaviour specification[8] and if *c* can cause *b* and it does not occur before *a*, *i.e.* (*c*,*b*) is in the transitive closure of the behaviour restricted to all those actions that follow *a*, then either *a* is equal to *c* or *a* can cause *c* also.

Establishing the set of actions between two others in a behaviour specification can be represented as:

$$allBetween : (action \times action \times behspec) \leftrightarrow \mathbb{F} action$$

$$\forall a; b : action;\ as : \mathbb{F} action;\ bs : behspec \mid ((a; b; bs); as) \in allBetween \bullet (\forall c : action \mid c \in as \bullet ((a; b; bs); c) \in isBetween)$$

Here the set of actions between two others in a behaviour specification is given by checking that individual actions are between these two actions. With this latter function it is possible to check that the number of actions between an invocation and termination does not exceed some upper limit. This can be represented as:

maxActionsBetween

$$\Xi OpIntTemplate$$
$$it? : InvTemplate$$
$$tt? : TermTemplate$$
$$s_1?; s_2?; s_3?; s_4? : State$$
$$maxActions : \mathbb{N}$$

$$\exists is : IntSig \mid is \in ops{:}ints \land it? = is{:}inv \land tt? \in is{:}terms \land (isInvAct(it?; s_1?; s_2?); isTermAct(tt?; s_3?; s_4?)) \in (opbs \rhd isInvAct(it?; s_1?; s_2?))^+ \bullet (\textbf{let}\ bA == max\ \{as : \mathbb{F} action \mid as = allBetween(isInvAct(it?; s_1?; s_2?); isTermAct(tt?; s_3?; s_4?); opbs) \bullet \#as\} \bullet maxActions \geq bA)$$

This is given an invocation template (*it*?), a termination template (*tt*?) and state information such that the invocation template, termination template and states are in the behaviour specification. The maximum number of actions between the invocation and termination is required to be less than some maximum value. We note here that we remove the actions that precede the invocation, *e.g.* the invocation might have been invoked previously and a termination given that was not associated with the invocation under consideration. For simplicity we assume that the invocation is not invoked again before the termination is issued. We discuss this specification fragment and the usage of Z to specify such issues at the end of this chapter.

6.3.3 Specifying Time Related Non-Functional Aspects in Z

Time-related aspects of behaviour may be represented in many ways in Z. We have seen in section 5.1.2 how certain timing aspects associated with multimedia flows could be represented. We consider now how Z might be used to reason about timed behaviour more generally.

In section 5.1.2 we modelled time through a natural number. As such we considered a discrete model of time. It is quite possible to model dense time though, *i.e.* as a real number. Real numbers do not exist in the Z mathematical toolkit, however, they can be modelled through the use of the toolkit. For example, reals could be introduced as a set $\mathbb{R}$ with operations defined over this set. This approach was taken in [186]. For simplicity here the following discussions treat time as a natural number.

Time can be used to restrict action occurrences in many ways. As with LOTOS, time itself can be modelled in different ways, *e.g.* local to processes or global throughout the specification. This distinction depends on the visibility and accessibility of the variable (or variables) modelling time. For example, time can be modelled as a variable that is accessed (and possibly changed) by all time-dependent operation schemas. Alternatively, different variables can be used to represent time. These variables can be used to

[8] For effective computation we remove all actions that occur following *b*.

model time local to operation schemas (or sets of operation schemas). Having a local model of time allows systems to be modelled where the components of the system can run at different speeds.

Once a model of time is chosen, this can then be used to restrict the behaviours of time-dependent actions (operation schemas). If an action is to be modelled so that it can only occur at the local time 3 say, then this can be specified as:

SomeOp

$$
\begin{array}{l}
\ldots \\
time : \mathbb{N} \\
\hline
\ldots \\
time = 3
\end{array}
$$

This operation schema does not itself model any modifications to the variable representing time. Indeed, as with the model of time in LOTOS given previously, it need not necessarily be the case in Z that time is modelled so that it progresses, *i.e.* operations may or may not increase the time. Typically we are interested in the temporal ordering of actions though and how actions occurring may prohibit (due to temporal considerations) other actions from occurring. It should also be pointed out that, given that we may model time as a natural number, it is quite possible to specify time as decreasing, *e.g.* through operation schemas that decrement variables modelling time. This would of course be meaningless(?).

In the previous specification fragment, the model of time was local to the operation schema. Modelling an action that could only occur at time 3 with a global time model might be represented as:

SomeOp

$$
\begin{array}{l}
\ldots \\
\hline
\ldots \\
time = 3
\end{array}
$$

Here the variable *time* might be represented as:

$$time == \mathbb{N}$$

It is quite possible to model time windows directly in Z. If an action can only occur within the global times $0..4$, then this can be represented as:

SomeOp

$$
\begin{array}{l}
\ldots \\
\hline
\ldots \\
0 \leq time \leq 4
\end{array}
$$

It is possible to model operation schemas so that they can only occur at regular time intervals. For example, an operation schema that can only occur at even global times, *i.e.* 0,2,4... can be represented as:

SomeOp

$$
\begin{array}{l}
\ldots \\
\hline
\ldots \\
time \bmod 2 \neq 1
\end{array}
$$

A potentially infinite variation of timing possibilities is possible through having different predicates that the variable modelling time should satisfy. This includes times at which the actions may not occur as well as times at which the action is allowed to occur.

In these examples so far, time itself has not been modified. As a result these models of actions can be regarded as instantaneous. It is quite possible to model actions with durations however. Actions with durations[9] require that the variable modelling time is modified. For example, an action that takes 2 local time units to complete can be represented as:

[9] or actions followed by time intervals.

SomeOp

$$\begin{array}{l} \ldots \\ t; t' : \mathbb{N} \\ \hline \ldots \\ t' = t + 2 \end{array}$$

Combinations of this model of actions with durations and actions with specific timing requirements can be specified directly. For example, an action that can only occur at time 3 and takes 5 time units to complete can be modelled as:

SomeOp

$$\begin{array}{l} \ldots \\ t; t' : \mathbb{N} \\ \hline \ldots \\ t = 3 \land t' = t + 5 \end{array}$$

In addition actions that may take different times to complete can be represented directly. For example, if the action modelled in the last specification fragment could last for 5 or 7 time units, then this could be represented by simply including a Boolean *or* in the final predicate, *i.e.* $t' = t + 5 \lor t' = t + 7$.

As can be seen, the Z language can be used to model timing issues directly. We show how these basic specification fragments for reasoning about time can be used to specify a producer and consumer flow configuration in chapter 8.

It is quite possible to model time in a more abstract fashion, *i.e.* modelling time with the abstract Z approach given in chapter 4. A basic model of a timed behaviour may be represented in Z by:

$$Tbehspec == \{tar : \mathbb{N} \times \mathbb{N} \times action \leftrightarrow \mathbb{N} \times \mathbb{N} \times action\}$$

Here we state that a timed behaviour specification is simply a relation between actions that have been time stamped. We consider here actions that have been time stamped with a start and end time. Variations are possible, *e.g.* only time stamping the time at which an action is allowed to occur (or before, or after). This model of a timed behaviour might be used to model timed operational interface templates as:

TimedOpIntTemplate

$$\begin{array}{l} \Xi OpIntTemplate \\ topbs : Tbehspec \\ \hline \forall t_1; t_2; t_3; t_4 : \mathbb{N};\ a_1; a_2 : action \mid t_1 \le t_2 \land t_3 \le t_4 \land t_1 \le t_3 \land \\ \qquad ((t_1; t_2; a_1); (t_3; t_4; a_2)) \in topbs \bullet (a_1; a_2) \in opbs \end{array}$$

Here we state that a timed behaviour does not expressly change the ordering of actions comprising the untimed behaviour of the operational interface template. If one action causes another action then it should have a less than or equal start time than the caused action. We also state that actions should have a start time that is less than or equal to their end time. We note that it is possible to model overlapping actions by allowing the start times of caused actions to be less than the end times of the actions that caused them. This in turn would require the simple model of behaviour as an action template ordering to be modified to allow for non-atomic action templates.

It is possible to adopt different policies with regard to how the time stamps themselves are associated with actions. Two examples of such policies are: an *ascending* policy and a *relative* policy. In an *ascending* policy, if one action can cause another then it should have an end time stamp that is less than or equal to the start time stamp of the caused action. If the time stamps are equal then this implies that the caused action should occur without time progressing, or it should not occur at all. Time stamping actions increases the constraints on those actions occurring. For example, if we have the behaviour $a \| b$ say then this may be modelled as $\{(a,b), (b,a)\}$. If these were time stamped as: $a_{\{5,7\}}; \|\, b_{\{2,4\}};$, then it is no longer the case that the behaviour specification can exhibit the behaviour *(a,b)*. The time for *b* to occur has passed if *a* occurs. Of course *a* may still follow *b* as it has a higher time stamp.

In a *relative* policy the time stamps are dependent upon the time difference between action occurrences. Thus the behaviour $b_{\{4,6\}};\ a_{\{7,9\}};$ using an ascending time stamp model might be represented by $b_{\{4,6\}};$

$a_{\{1,3\}}$; using a relative policy. This latter approach is the more commonly taken approach in timed process algebras, *e.g.* T-LOTOS [139].

These different time stamping policies place different requirements on the checks for timing issues. We consider now how it is possible to model the maximum (or minimum) times from the occurrence of an invocation to the occurrence of an associated termination with these two time stamping policies.

Maximum Time Between Invocation to Termination with Ascending Policy

It is trivial to establish the maximum time difference between an invocation and one of its associated terminations in a timed behaviour specification with actions time stamped in ascending order. It is simply represented by the maximum end time stamp associated with one of the terminations associated with the invocation. This can be represented by:

maxTimeInvTermAscendingPolicy

$\Xi TimedOpIntTemplate$
$it? : InvTemplate$
$tt? : TermTemplate$
$s_1?; s_2?; s_3?; s_4? : State$
$tnow?; tmax : \mathbb{N}$

$\exists is : IntSig \mid is \in ops{:}ints \land it? = is{:}inv \land tt? \in is{:}terms \bullet$
$tmax = max\ \{t_1; t_2; t_3 : \mathbb{N} \mid$
$((tnow?; t_1; isInvAct(it; s_1?; s_2?)); (t_2; t_3; isTermAct(tt; s_3?; s_4?))) \in$
$(topbs\ \{isInvAct(it; s_1?; s_2?)\})^+ \bullet t_3\} - tnow?$

Here we state that the maximum time from an invocation to termination is given by the maximum end time stamp associated with one of the associated terminations minus the current time. The minimum time from an invocation to a termination may be obtained by changing the predicate here to return the minimum of the set. For brevity, we omit the Z text for this. We note here also that we do not consider terminations associated with previous invocations. For simplicity we assume that the invocation is only invoked once before the termination is sent.

Maximum Time Between Invocation to Termination with Relative Policy

If actions were time stamped in relation to one another then establishing the maximum time from an invocation to a termination requires summing all time stamps of actions in the sequence of actions between the invocation to the termination. The maximum time is the largest of these sums. To specify this we first need to introduce two functions. The first function takes a sequence of timed actions and returns the sum of the times associated with these actions. This can be represented by:

$SumTimes : seq(\mathbb{N} \times \mathbb{N} \times action) \rightarrow \mathbb{N}$

$\forall sta : seq(\mathbb{N} \times \mathbb{N} \times action) \bullet$
$sta = \langle\rangle \Rightarrow SumTimes(sta) = 0 \lor$
$sta \neq \langle\rangle \Rightarrow SumTimes(sta) = Second\ (head\ sta) + SumTimes(tail\ sta)$

We note here that in a relative policy, the sum of the times is given by the times at which the actions end. For example, if the actions with an ascending policy were $a_{\{1; 2\}}$; $b_{\{4; 6\}}$; $c_{\{8; 10\}}$ then this would be represented as $a_{\{1; 2\}}$; $b_{\{2; 4\}}$; $c_{\{2; 4\}}$. Thus c ends at (2+4+4=10), *i.e.* the sum of the end times of the actions. For brevity we do not provide the specification text for the function *Second*. For the Z mathematical toolkit provides two operations on tuples: first and second. These operate on pairs of values. The function *Second* here returns the second element of a triple.

The second function takes a timed action sequence and returns the untimed action sequence. This can be represented as:

$UntimeSequence : seq(\mathbb{N} \times \mathbb{N} \times action) \rightarrow seq\ action$

$\forall sta : seq(\mathbb{N} \times \mathbb{N} \times action) \bullet$
$sta = \langle\rangle \Rightarrow UntimeSequence(sta) = \langle\rangle$
$sta \neq \langle\rangle \Rightarrow UntimeSequence(sta) = \langle Third\ head\ (sta)\rangle \frown tail\ sta$

We omit the Z text for *Third* for brevity. This returns the third element of a triple.

The maximum time between the occurrence of an invocation to the occurrence of one of its associated terminations where actions are time stamped in relation to one another may be represented as:

maxTimeInvTermDifferencePolicy

$\Xi OpIntTemplate$
$it? : InvTemplate$
$tt? : TermTemplate$
$s_1?; s_2?; s_3?; s_4? : State$
$tmax : \mathbb{N}$

$\exists is : IntSig \mid is \in ops{:}ints \land it? = is{:}inv \land tt? \in is{:}terms \bullet$
$tmax = max \{sta : seq(\mathbb{N} \times \mathbb{N} \times action); sa : seq\ action \mid$
$\quad head\ sa = isInvAct(it?; s_1?; s_2?) \land last\ sa = isTermAct(tt?; s_3?; s_4?) \land$
$\quad UntimeSequence(sta) = sa \land$
$\quad ran\ sa \subseteq allBetween(isInvAct(it?; s_1?; s_2?); isTermAct(tt?; s_3?; s_4?); opbs) \bullet SumTimes(sta)\}$

Here we are given an invocation and termination template associated with some interrogation signature. The maximum time between the invocation and termination (with a relative time stamping policy) is given building sequences of actions that are started with the invocation and end with the termination, and only contain actions found between the invocation and termination in the behaviour specification. The maximum time is obtained by summing the end time stamps for all of these possible sequences and returning the maximum value. We note here that if the invocation did not start at time zero, then it would be necessary to subtract the time at which it was invoked from this final value. We discuss the approach used in this schema at the end of this chapter.

6.3.4 Specifying Aspects of Multimedia in Z

We consider now how Z can be used to model timing issues related to multimedia flows of information. We focus here on upper and lower delay limits of information flows and throughput considerations of information flows. We shall see in chapter 8 how the approach developed so far can be used to reason about other aspects of information flows, *e.g.* bound and unbound jitter flows and latency. In section 5.1.2 we presented an elementary model of a multimedia flow of information. This was given as:

mmFlowType

$frames : seq\ Frame$
$flowChar : FlowChar$
$rate : \mathbb{N}$

$:::$

This model provided a basic outline of the features inherent to multimedia flows of information. It is possible to extend this basic model to allow assertions to be made on the information flows also. We shall see in the following chapter that these assertions provide a basis for determining whether interfaces can be composed with one another and function correctly.

Maximum and Minimum Delay between Frames in Z

The maximum delay of a multimedia flow may be regarded as the upper limit on the time window at which a frame is expected. For example, a consumer may be able to wait for a certain time for the next frame to arrive. Without considering issues such as buffering, the maximum delay associated with an information flow may be represented as:

$maxDelay : mmFlowType \rightarrow \mathbb{N}$

$\forall mft : mmFlowType \bullet$
$\quad mft{:}flowChar = Isoch(mft{:}rate) \Rightarrow maxDelay(mft) = mft{:}rate \land$
$\quad mft{:}flowChar = Bursty \Rightarrow$
$\quad\quad maxDelay(mft) = max \{f_1; f_2 : Frame \mid \langle f_1; f_2 \rangle in\ mft{:}frames \bullet f_2{:}timestamp - f_1{:}timestamp\}$

For isochronous flows, the maximum delay that a consumer can tolerate without buffering is given by the time difference between two frames. That is, if after this time period the frame is not received, then an error has occurred and some remedying action must be taken, *e.g.* show last frame again. For bursty flows, the maximum delay is given by the maximum time difference between two successive frames in the sequence, *i.e.* if the consumer can consume as fast as the producer can produce, then consumption should be at least as fast as production.

The minimum delay of a multimedia flow may be regarded as the lower limit on the time window at which a frame is expected. As with maximum delay constraints, for isochronous flows the minimum delay corresponds simply to the rate. For bursty flows though, the minimum delay is the minimum time difference between two successive frames in the sequence. Thus the final predicate should be modified to return the minimum time difference between two frames. We omit the Z text for brevity.

Maximum and Minimum Throughput of Flows in Z

Throughput may be regarded as the number of frames that a producer of a flow wishes to produce, or the number of frames that a consumer wishes to consume. Isochronous flows should have a consistent throughput hence their maximum and minimum throughputs should be the same. Bursty flows may have situations where more frames are output (or input) than at other times. In bursty flows it is especially useful to put an upper limit on the maximum throughputs of the data. Thus throughput may be represented as:

$$
\begin{array}{|l}
maxThruPut : mmFlowType \rightarrow \mathbb{N} \\
\hline
\forall mft : mmFlowType \bullet \\
\quad mft.flowChar = Isoch(mft.rate) \Rightarrow maxThruPut(mft) = 1 \operatorname{div} mft.rate \wedge \\
\quad mft.flowChar = Bursty \Rightarrow \\
\quad\quad maxThruPut(mft) = max \{ s : \operatorname{seq} Frame; f_1; f_2 : Frame \mid s \operatorname{in} mft.frames \wedge \\
\quad\quad\quad f_1 = head\ s \wedge f_2 = last\ s \wedge f_2.timestamp - f_1.timestamp \leq 1 \bullet \#s \}
\end{array}
$$

Here the throughput of isochronous flows is simply represented by the reciprocal of the rate, *i.e.* if the time difference between successive frames was 0.1 seconds then the throughput would be 10. Establishing the throughput of bursty flows is a little more involved however. Here the maximum throughput of a bursty flow is obtained by calculating the maximum subsequence of the flow with a timestamp difference of less than or equal to one second from its first and last elements.

As stated in section 6.2.4, throughput is typically measured over a period of time as opposed to a fixed time unit (here 1). This may be modelled as:

$$
\begin{array}{|l}
maxThruPut : mmFlowType \times \mathbb{N} \rightarrow \mathbb{N} \\
\hline
\forall mft : mmFlowType \bullet \\
\quad mft.flowChar = Isoch(mft.rate) \Rightarrow maxThruPut(mft; n) = n \operatorname{div} mft.rate \wedge \\
\quad mft.flowChar = Bursty \Rightarrow \\
\quad\quad maxThruPut(mft; n) = max \{ s : \operatorname{seq} Frame; f_1; f_2 : Frame \mid s \operatorname{in} mft.frames \wedge \\
\quad\quad\quad f_1 = head\ s \wedge f_2 = last\ s \wedge f_2.timestamp - f_1.timestamp \leq n \bullet \#s \}
\end{array}
$$

Here the throughput of isochronous flows is simply represented by the time over which the throughput is measured divided by the rate. The maximum throughput of bursty flows is obtained by calculating the maximum subsequence of the flow with first and last elements having a time difference less than or equal to the time period over which the throughput was measured

The minimum throuput of a bursty flow may be obtained simply by changing the final predicates of these axiomatic descriptions so that they return the minimum sequence of frames. For brevity, we omit the Z text.

These specification fragments may be regarded as assertions on the flows themselves. We shall see in the following chapter how these assertions may be used to compose systems, and in chapter 8 how these assertions can themselves be checked.

6.4 Summary

In this chapter we have shown how LOTOS and Z can be used to specify a set of non-functional aspects (constraints) of systems. We have focused here on timing constraints, resource usage constraints and cost constraints, although as stated it is quite possible to specify numerous other non-functional aspects or constraints.

In LOTOS we have specified the constraints in Act One and used these as a basis for limiting the possible behaviours that can occur in the process algebra. Several issues result from the specification of resource usage and timing considerations in particular. Our model of resource usage was based on keeping a count of the actions that have occurred, *i.e.* actions that start new activities (invocations) and actions that stop existing activities (terminations). This model may only be regarded as a rough approximation at best to resource usage. That is, this model treats all actions as being equal consumers of resources. It is quite possible to extend the approach so that different actions consume more resources than others say. This requires a more detailed model of behaviour than has been given here, *i.e.* we do not work at the level of dealing with explicit examples of behaviours.

We have focused on a model of time in LOTOS with particular emphasis on its application to information flows. The approach given has shown the flexibility and power of the LOTOS language. It is not the case that languages for describing real time systems should necessarily have built-in features for representing time explicitly. The approach given here for specifying and reasoning about time in LOTOS is not the only one. For example, issues such as jitter and time windows for action occurrences could be engineered into the Act One sorts modelling the operations (actions) themselves, as opposed to simple fields for time stamping.

The approach given is not without its problems, especially with regard to the semantics of LOTOS. Modelling actions with durations can be achieved abstractly, but the inherent semantics of LOTOS is that actions are atomic. Thus, we do not have a true concurrency model for our timed extension, *i.e.* where actions can overlap in time. Despite this, it is possible to specify and reason about a wide range of time-dependent systems.

The Z specification fragments in this chapter have shown that Z is particularly suited to specifying non-functional aspects of systems. Since everything in Z is a set, satisfaction of these constraints is given by set membership. This set membership may well have predicates associated with it. We have seen that timing issues can be represented in Z in many diverse ways. We shall consider the timing demands of multimedia systems in chapter 8 when we specify a producer and consumer flow configuration.

Z is an exceptionally expressive language. This has both advantages and disadvantages. If we consider the purist view that a specification denotes what a system is supposed to do, as opposed to how it is supposed to do it, then Z is a very powerful and useful specification language. For example, the specification fragments given previously to capture the maximum number of actions between an invocation and termination, or to capture the maximum times between invocations and terminations, highlight the power of the language. Such expressiveness is not without its problems. That is, whilst it is possible to specify that the maximum time between an invocation and termination should not exceed t time units, say, ensuring that this is the case is another matter.

There are two main ways in which this problem could be addressed. Refinement could be used to iteratively produce a less abstract specification, *i.e.* a specification that was closer to an implementation. Typically refinement is a laborious (and difficult) procedure, where many proofs are required. With a particularly abstract and complex specification, such approaches are not always viable. An alternatively approach is to develop some form of operational semantics in Z similar to that found in LOTOS, say, and use this as a basis to ensure that the abstract (expressive) statements about behaviour are fulfilled. Such an approach (whilst possible) to some extent goes against the fundamental Z approach. That is, if something operational is required then the question might well be asked as to why not use LOTOS in the first place. The same argument might equally well be put forward against the abstract Z approach given here. We argue that our approach allows aspects of behaviour to be captured in a manner that would not be possible in a less abstract specification technique.

Chapter 7

Type Checking in Distributed Systems

In this chapter we argue that types and type checking generally are the most fundamental constructs for developing distributed systems and, importantly with regard to this thesis, architecting specifications of distributed systems. We begin with an overview of type checking as it has traditionally been conceived and introduce the special requirements that distributed systems impose on type systems. Our focus is predominantly on interface types and how they may be used to compose sub-systems or substitute one system for another.

Types and type checking approaches are then considered in LOTOS. The limitations of the language for modelling interface types in a form that allows for their checking are shown, and an approach is provided that allows interface signature checking to be achieved. As discussed in chapter 1 though, signature compatibility of types lends itself to a "message is understood" policy only. Whilst useful as a basis for considering interworking between systems, it is only a basis. We then proceed to show how the work in chapters 5 and 6 can be used to extend type checking approaches based on signature checking only.

As well as the problems of treating typing as a signature check only, we identify other problems with the ODP type system. These are the lack of distinction between clients and servers and the special requirements they place on type checking issues with regard to binding and substitution. A specification of the necessary conditions for safe binding and substitutability in an open distributed system is then provided in LOTOS and Z.

7.1 Types and Type Checking Traditionally

Typing is one of the most fundamental concepts in computing science. From a low level programming point of view it offers two main features, namely data modelling and data protection. Blair notes [15] that this data modelling aspect fulfils two related roles:

- It enables programmers to build abstractions to overcome the underlying properties of entities in the system. For example this includes the ability to abstract over how entities, *e.g.* integers, are stored and represented by the computer at the memory level.
- It offers a means to develop higher level abstractions from existing abstractions. For example, basic types such as Booleans and integers may be used to build more complex (aggregate) types such as records, which might in turn be used to build even more complex types. This abstraction building is an essential weapon in the programmer's armoury.

In this chapter it is primarily issues with regard to data consistency that we investigate, specifically on the notion of type equivalence. What does it mean for two types to be regarded as being equal?

Type equivalence historically can be largely divided into two approaches: type checking by name matching and type checking by structural equivalence. In name equivalence, two values have equivalent types if the types share the same declaration, *e.g. x, y: A* declares *x* and *y* to be of the same type, namely *A*. Structural equivalence in its strictest sense, says that two types are equivalent if the types have isomorphic

structures. We shall see presently how differing type structures or similar structures with different naming schemes also bring interesting and useful type relationships in section. We highlight the differences between name and (strict) structural equivalence through an example using the Ada language [26]:

```
type Person is Record            type Car is Record
      age: Integer;                   age: Integer;
      weight: Real;                   weight: Real;
endrecord;                          endrecord;
```

In structural terms, Person and Car define the same set of values *i.e.* the cross product *age: Integer* $\times$ *weight: Real*. However in name equivalence terms there is no relation between Person and Car. There are advantages and disadvantages with this. Perhaps the biggest advantage of name equivalence is its simplicity. Checking type equivalence in this way corresponds to checking whether two strings are the same. This advantage goes a long way to explaining why name equivalence is such a popular scheme within programming languages, *e.g.* Ada [26].

Name equivalence is not without its drawbacks however. One disadvantage of name equivalence is that it does not allow for anonymous types. For example, one cannot write procedures that accept parameters of a certain shape, *e.g.*

```
procedure Dummy (X: array(1..10) of Integer);
```

Rather, type definitions have to be used to resolve the problem, *e.g.*

```
type IntArray is array (1..10) of Integer;
procedure Dummy(X: IntArray);
```

A second and much more serious problem with name equivalence is its restriction to subtype checking. Type equivalence on name equivalence only prohibits subtyping as used for inclusion polymorphism in object-oriented languages. For example, consider the following type:

```
type WorkingPerson is record
     age: Integer;
     weight: Real;
     salary: Real;
endrecord;
```

Cardelli and Wegner [40] state that this is a subtype of Person since it has all of the same attributes (and more). There are two ways in which this subtype relationship may be checked: implicitly whereby subtypes are established by the type system itself, or explicitly whereby the user declares the type relationships themselves, *e.g.*:

```
type WorkingPerson is a subtype of Person;
```

This information may then be used to establish different name equivalence relationships. In an open distributed system however, such user information is unlikely to always be available. System evolution means that new resources may be found and be expected to interwork with existing components. Given the possibility of autonomous subsystems, the likelihood of users being able to assert such relationships is remote.

A second approach to determining the relationships between Person and Car (and Person and WorkingPerson) is through structural equivalence. Structural equivalence of types poses a different set of requirements on establishing type equivalence. There are, as with name equivalence, advantages and disadvantages to type checking based on structural equivalence. One disadvantage is that, in the main, type checking based on structural equivalence is more complex than name equivalence and hence requires more processing time. Perhaps the greatest advantage of type checking based on structural equivalence is that it lends itself to polymorphism as discussed in section 2.2.2.

We note here though that name equivalence and structural equivalence when used in conjunction offer perhaps the most flexible of approaches. That is, whilst it will not always be possible to assert that one type is a subtype of another type, in certain instances this might be possible. For example, assertions on the more common types can be made that can aid in establishing type relationships, *e.g.* integer is a subtype of

real. From this, establishing other relationships is made possible, as well as determining more efficiently whether type relationships exist or not. Thus if it was known that integers could be substituted for reals, *i.e.* coercions between the two types were possible, then the types Car and Person given earlier could be substituted[1] for the following type:

```
type Vehicle is record
     age: Real;
     weight: Real;
endrecord;
```

The importance of the labels for types also needs to be addressed when type checking is done. For example, few type systems can establish a relationship between *Vehicle* and *Lorry* due to the operation names associated with the types being different:

```
type Lorry is record
     lorryAge: Real;
     lorryWeight: Real;
endrecord;
```

Automated syntactic structural type equivalence then, in the general case of arbitrary type structures without a common name basis, is not possible without user intervention. Nevertheless establishing the structural form of the type can aid in establishing where type relationships do not exist.

We now investigate how types are represented in LOTOS and the limitations of LOTOS in its classical usage for specifying type systems in a checkable manner.

7.2 Types in LOTOS

Types may be modelled in LOTOS in two main ways. They can be represented in Act One directly, or as shown in section 4.3.18 indirectly in the process algebra. As we shall see, both of these approaches are limited with regard to the possible type checking requirements that might be found in a distributed system.

7.2.1 Process Algebra Types

It is not strictly the case that the process algebra part of LOTOS is used to represent types. The process algebra models behaviour through processes and behaviour expressions. As discussed in section 4.3.7, an ODP type as a characterising predicate cannot be formalised in LOTOS (or Z) due to its generality. ODP is more prescriptive in its definition of template types though. A template type (see section 4.3.18) is a predicate defined in a template that holds for all instantiations of that template and that expresses the requirements that instantiations of the template are expected to fulfill.

Given this definition, a template type in the process algebra corresponds to an instantiation of a process definition. Any LOTOS specification that has a correct static semantics, *i.e.* all process definition instantiations in the specification satisfy the LOTOS static semantics rules given in [101], satisfy the template type. Whilst correct, this interpretation of typing through template typing and the LOTOS static semantics is limiting since it prohibits subtype checking. As discussed previously in section 4.3.9, a subtype should satisfy the same predicate as the supertype. In the process algebra, it is not possible to reason about types and subtypes as first class entities. We illustrate this with an example. Consider the following fragment of LOTOS consisting of two template types, *i.e.* the two process definitions and their associated instantiations:

```
      Temp1[g1,g2] ||| Temp2[g2]
where
      process Temp1[g1,g2]:exit:=       process Temp2[g2]:exit:=
          g1; exit [] Temp2[g2]            g2; exit
      endproc (* Temp1 *)               endproc (* Temp2 *)
```

[1] We shall discuss this area of substitutability shortly, since this substitution may not always be possible when causality is considered.

Here the template type of process definition *Temp1* and its instantiation also satisfies the predicate associated with the template type of process definition *Temp2* and its instantiation. This is because process definition *Temp2* makes up one part of the behaviour expression of *Temp1* and the instantiations of both processes are compatible. That is, the part of the behaviour expressions that *Temp1* and *Temp2* have in common are both instantiated with gate *g2*. Thus in an environment expecting *Temp2* instantiated with gate *g2*, an instantiation of *Temp1* with gate *g2* as the second gate, would suffice.

Although this method of defining types and subtypes is valid it is not particularly useful. It is not possible using this type/subtype model to return the subtypes of a given type, to list the properties of a type, or even to check that two types are in a subtyping relation within the constraints of LOTOS. That is the specifier can see (or prove) that *Temp1* is a subtype of *Temp2* but the specification cannot derive this fact for itself.

Thus with this modelling of types, any type can synchronise with another type even if they are different types. That is, there exists no feature in LOTOS which checks that the type of process definition instantiation *X* is the same as process definition instantiation *Y* and therefore they can synchronise. In LOTOS any process definition instantiations synchronised by the parallel composition operators $\|$ or $|$ [*:::*] $|$ can interact in some way with each other and it is up to the specifier to ensure that they have the correct types, *i.e.* following synchronisation the desired behaviour (or any behaviour) will occur. Undesired behaviour might manifest itself as deadlock in the specification.

An alternative to modelling types and subtypes as instantiations of process definitions is to model them through ADTs in the Act One part of LOTOS.

7.2.2 The Abstract Data Type Approach

The ADT approach is to model types[2] as instances of sorts in the Act One part of LOTOS. Consider the ADT *X_type* given here:

```
type X_type is
     sorts X
     opns a: -> X
          b: X -> X
          c: X,X -> X
endtype (* X_type *)
```

```
type Y_type is
     sorts X, Y
     opns a: -> X
          b: X -> X
          c: X,X -> X
          d: X -> Y
endtype (* Y_type *)
```

In the behaviour part of LOTOS, when a value is declared of sort *X* then its sort defines the predicate. For example, if we have *x: X* declared then the predicate satisfied by *x* is that it has the operations *a,b* and *c* and satisfies the equations (if any are given)[3].

The problem with this approach as far as a type management system is concerned, however, is that it is not possible to establish subtyping relationships or type equivalence in a form that is checkable, *i.e.* in a form that is not based entirely on trust. For example, it would be desirable to have an operation of the kind *is_subtype: Y, X* $\rightarrow$ *Bool*. Before going on to state why this is the case it is necessary first to establish formally what exactly is meant by type equivalence and subtyping in ADTs. We shall see in section 7.3 that these basic ideas do not always hold true when causality is introduced, *i.e.* when the client-server paradigm is used.

7.2.3 Typing and Subtyping in ADTs

Type equality may be interpreted in terms of subtyping. Thus if $\leq$ is given as the subtyping relation between two ADTs *adt1* and *adt2* say, then *adt1* equals *adt2* if they are both subtypes of each other. That is:

$$adt1 \leq adt2 \wedge adt2 \leq adt1$$

The obvious question now arises as to what exactly is subtyping between ADTs in Act One. Given two ADT signatures *adt1* and *adt2*, then *adt1* is a subtype of *adt2* provided the operations of *adt2* are

[2] We note here that a LOTOS type is distinct from a type as might be found elsewhere. That is, types in LOTOS are more structuring units for grouping sorts, operations and equations together.

[3] If no equations are given then we have only a signature type, as opposed to full behavioural subtyping.

contained in or are acceptable substitutes for the operations of *adt1*. Here acceptible substitute implies that the input and result parameters of the operations are in appropriate subtyping relations. Intuitively, with this interpretation of subtyping, a subtype should allow more inputs and produce less outputs. For simplicity we assume that the parameters are identical and do not concern ourselves with issues of contra-variance and co-variance.

For any ADTs for which subtyping is expected to hold this implies:

1. all operation names in *adt2* have corresponding operation names in *adt1* (corresponding in the sense of representing the same operation);

2. the common operations have the same number and type (sorts) of input parameters (arguments);

3. the common operations have the same prefix/infix nature;

4. the common operations produce the same result types (sorts)[4].

As identified in [15], this is a restrictive subtyping relation in that all of the operations of the supertype have to be included in the subtype, even if they will not be invoked in a given context.

Consider now the ADT *Y_type* given above. >From the type rules given above, one would expect that *Y_type* was a subtype of *X_type*. However, LOTOS establishes types through instances of sorts. It is not meaningful in LOTOS to declare something like *y: Y_type* or *x: X_type*. As stated previously, the type constructs of LOTOS merely provide a means to collect together sorts and operations (and equations). The only types we can declare in the process algebra are instances of sorts, *i.e. x: X* and *y: Y*. The successful synchronisation of event offers requires that they have compatible action denotations. Compatibility here requires that the parameters in the event offers are declared as being instances of the same sort with intersecting values.

>From this it can be gathered that LOTOS uses name equivalence when type checking is done. That is, two types are the same only when they have the same name, *i.e. x: X* is only equal to some other value *a: A* when *A* sort is in fact an *X* sort.

This is a serious restriction on the usefulness of ADTs to specify "types" directly, as it limits the checking that can be performed. Specifically, here we are restricted to name equivalence. Ideally, checks on structural similarity should be made possible, where structural similarity is similarly defined to subtyping of ADT signatures with looser constraints on the inputs and result parameters. Specifically, rules 2 and 4 of the subtyping rules given above should be modified to:

- the common operations have the same number of input parameters and these must be used consistently throughout the entire signature;

- the common operations produce the same result types (sorts) and these must be used consistently throughout the entire signature.

This consistent usage of a sort in a signature means that two ADTs using different sorts (labels) are structurally similar only if the sorts in the ADTs signatures can be relabelled consistently throughout the signature so that the corresponding operations they share are identical. If the corresponding operations they share represent the whole signatures of the ADTs, then the structural similarity the ADTs share is in fact a structural equivalence. Another way of considering this is that if one ADT is structurally similar to a second ADT, and the second ADT is structurally similar to the first, then they are structurally equivalent. Thus in effect, structural equivalence corresponds to type equivalence, and structural similarity corresponds to subtyping provided it can be established by some means (see 7.2.6) that the sorts in the signatures are semantically equivalent. For simplicity we do not consider extensions to these rules to take account of different numbers of inputs, *e.g.* inputs of the form *makeNatPair: Nat, Nat → NatPair* which would allow *NatPair* to be used where two natural numbers were expected say. As an example of these relationships consider the following ADT signatures:

[4] In LOTOS there is no need to consider the number of result types, as Act One operations only return one result.

```
type A_type is
     sorts A
     opns a: -> A
          b: A -> A
          c: A,A -> A
endtype (* A_type *)
```

```
type B_type is
     sorts B
     opns a: -> B
          b: B,B -> B
          c: B,B -> B
endtype (* B_type *)
```

```
type C_type is
     sorts C
     opns a: -> C
          b: C -> C
          c: C,C -> C
          d: C,C,C -> C
endtype (* C_type *)
```

Here, *A_type* is structurally equivalent to *X_type* given above, as relabelling the *A* label with an *X* label makes the *A_type* signature identical to the *X_type* signature. *X_type* is structurally similar to *C_type* as relabelling the *C* label (sort) by an *X* label in the *C_type* signature makes the corresponding operations they share (*a,b,c*) identical. However, *X_type* has no immediately identifiable structural relationship to *B_type* as operation *b* requires too many input parameters, hence relabelling will not satisfy the above rules for structural similarity.

In fact in this example there exists a direct isomorphism between ADTs *A_type* and *X_type*. That is, they differ only in the label that is attached to the carrier in the signature, *i.e.* the sort. Thus a bijection exists from *A_type* to *X_type* (and hence an inverse relation) which is achieved by relabelling the sort (*X* or *A*) in the two ADT signatures. It is worth noting that in homogeneous algebras, *i.e.* algebras with a single sort as given here in *A_type*, rule 2 of the subtyping rules requires that only the number of parameters of corresponding operations is the same as relabelling will always be consistent. In heterogeneous algebras, *i.e.* algebras with more than a single sort as in *Y_type* given above, the number and sorts of parameters and results need to be considered for corresponding operations. Thus for example, given the following ADTs:

```
type P_type is
     sorts P, Q
     opns a: -> P
          b: P -> P
          c: P,P -> P
          d: P -> Q
endtype  (* P_type *)
```

```
type R_type is
     sorts R, S
     opns a: -> R
          b: R -> R
          c: R,R -> R
          d: S -> R
endtype (* R_type *)
```

P_type is structurally equivalent to *Y_type* above as relabelling *X* and *Y* in the signature of the ADT *Y_type* by labels *P* and *Q* respectively would give the same signature as ADT *P_type*. However, this is not the case with *Y_type* and *R_type*, *i.e.* relabelling *X* and *Y* by labels *R* and *S* respectively, as this would result in operation *d* having *R* as input and *S* as output, which is not the same as ADT *R_type*.

It should be pointed out that being able to identify structural similarity also enables browsing of types to be facilitated. For example, requests of the form, 'I know what structure (or partial structure) for a type I want but I don't know the name', could be satisfied. Augmenting structural similarity checks with name checking corresponds to full signature type checking. In this case checks can be performed to ensure that a given type has the correct name and structure associated with that type.

7.2.4 The Failure of the Direct ADT Approach

Having identified the different forms of type checking possible and noted that establishing structural equivalence is a desirable property for a type management system to possess; the question now arises as to how structural equivalence can be established in LOTOS. That is, how can it be established that *x: X* is structurally equivalent to *a: A*? The specifier can see immediately that this is the case, but the specification cannot derive this fact for itself. To establish structural equivalence would require something of the form:

```
all_ops(X) -> Op_Set
```

which would return the operation set $\{a; b; c\}$ (with the associated input and output parameters for each operation) for X and for A. It would be checked that the operations on these sorts have the same names, the same number of inputs and differ only in the label that is attached to the sorts throughout the signature. Structural equivalence could then be checked by an operation of the form:

```
is_se(X,A) -> is_eq(all_ops(relabel(X,A)),all_ops(A))
```

where *is_se* checks for structural equivalence, and *all_ops*, *is_eq* and *relabel* are meant to be obvious. The problem is, however, LOTOS cannot have an operation like *all_ops*. That is, it cannot return a set of the operations associated with a given sort. Hence Act One is limited in its classical usage for establishing type relationships other than those based on name equivalence.

7.2.5 Establishing Structural Similarities in Act One

>From the above arguments it is obvious that the first two methods of representing types and subtypes in LOTOS have serious limitations in the checks that can be performed when type checking. One solution to establishing signature compatibility between types in LOTOS is through Act One describing Act One. That is, the simplistic idea of subtyping as a comparison between instances of sorts and their operations is lacking only in a means to compare instances, *i.e.* we need to be able to model the operation *all_ops* to enable structural equivalence relationships between types to be checked. Thus when provided with a naming scheme and in possession of a system capable of checking structural equivalence, type (and subtype) checking may be enforced.

As shown above, when given types like *X_type* and *A_type* we would like to be able to determine whether they are structurally equivalent to one another. This requires checking that they possess corresponding operations. In LOTOS an operation may be given by the quadruple:

```
Operation = < Operation_Name, In_Pre_fix, Input_Sorts, Output_Sort >
```

where the Operation_Name is simply an identifier (not necessarily unique), the In_Pre_fix denotes whether the operation is of prefix or infix nature, the Input_Sorts are a list of sort names (possibly an empty list as in operation *a* from ADT *X* above), and the Output_Sort is a sort name.

To compare whether two operations are the same in LOTOS it is necessary to check that they have compatible operation quadruples, *i.e.* same names, same prefix/infix notation, and compatible inputs and results. Comparison of operation names is trivial in LOTOS as it reduces to comparison of identifiers. Checking the infix or prefix nature of operations may be achieved simply by ensuring that when an operation is defined in LOTOS, a Boolean is given that determines whether the operation is of prefix or infix nature. Checking compatible input and output sorts may be achieved by comparison of identifiers. In a given naming scheme, all identifiers (sorts) in the signature will be known. Hence checking compatible input and output sorts can be achieved directly.

Using this approach of breaking down an operation into separate components, it is necessary to have some method of putting the components together again in a form that will be understood by LOTOS. We achieve this using the operation *make_op*. This is given by:

```
make_op : Op_Name, In_Pre_fix, Sort_List, ResSort -> Op
```

where an *Op_Name* is simply an identifier, an *In_Pre_fix* identifier is a Boolean denoting whether the operation is infix or prefix, a *Sort_List* is a list of identifiers, a *Sort* is an identifier, and an *Op* represents the operation. Thus equality of operations may now be checked by:

```
op1 eq op2 = ((get_name(op1) eq get_name(op2)) and
              (get-inpre(op1) eq get-inpre(op2)) and
              (is_eq_list(get_inputs(op1),get_inputs(op2))) and
              (is_eq_sort(get_output(op1),get_output(op2))));
```

where

```
get_name(make_op(opn1,inpre,sl1,s2)) = opn1;
get-inpre(make_op(opn1,inpre,sl1,s2)) = inpre;
get_inputs(make_op(opn1,inpre,sl1,s2)) = sl1;
get_output(make_op(opn1,inpre,sl1,s2)) = s2;
```

and operation *is_eq_list* checks whether two lists are the same and returns a Boolean result, *is_eq_sort* checks whether two sorts are the same and returns a Boolean result. Having a method of determining whether two operations are the same reduces the problem of subtyping between ADTs to a trivial set comparison, where set elements are the created operations.

With this approach, it is necessary to create the types explicitly. Thus if ADT *X_type* given above were to be created, this would be given by:

```
sort(X) = all_sorts(X,{})
ops(X)  = Insert(make_op(a,prefix,<>,X),
          Insert(make_op(b,prefix,add_sort(X,<>),X),
          Insert(make_op(c,prefix,add_sort(X,add_sort(X,<>)), X), {})))
```

where <> gives an empty *Sort_List*, {} gives empty sets to contain the operations and sorts of the signature, *all_sorts* adds the sorts contained in the signature into a set, *add_sort* inserts an identifier into a list, *prefix* declares the operation to be of prefix nature and *Insert* adds an element to a set.

Thus subtyping between two types, *y: Y_type* and *x: X_type* could be checked by the operation:

```
y isSubtype x = ops(y) Includes ops(x);
```

where *Includes* checks whether one set is a subset of another set. We shall see that this model of subtyping through operation subsetting does not always hold when issues of causality are considered.

7.2.6 Type Naming Issues

Management of types in a distributed system is crucial for their openness and evolutionary nature [116]. A complete specification of a type management system that incorporates many of the ideas given here for estabishing type relationships is given in Appendix A. This specification, as well as being able to determine whether subtyping between types exists, allows type relationships to be asserted and checks to be made on the truthfulness of these assertions with regard to structural aspects of the types.

As stated previously, this approach permits signature checking of types. The semantics of types occurs in this approach only when a suitable type naming scheme exists. It might also be possible to apply the same approach to the equations of the ADTs, that is, reduce them to a string compatibility check. However, this is more difficult since there are many ways of representing equations in Act One. An alternative to dealing explicitly with the behavioural aspects of types is to be more selective with regard to naming issues. As given, the ADTs specified and subsequently checked represent an empty shell with no semantics other than that suggested by the operation names. Although *X_type* and *A_type* are type equivalent up to isomorphism, their semantics might be entirely different. For example, label *X* might represent natural numbers and label *A* Booleans. Hence *a,b,c* might represent the operations “0”, “succ” and “+” for X, and “true”, “not”, “or” for A respectively. Therefore they can only be considered type equivalent if the labels (sorts) used in the signatures can be established as being the same semantically. This might be done via human intervention. That is, when a new type is supplied to the type management system, its name is supplied with its associated semantics. The problem of determining when two types are equivalent then becomes a naming issue. This does not imply that the type management system is reduced to “only” name equivalencing in its type checking, however. A type management system should perform both name equivalencing and structural equivalence when type checking.

This raises several issues which have to be addressed. For example, considerable trust is required to ensure that the types registered with the type management system are unambiguous and have the semantics as expected. This might be solved through an engineering aproach. For instance, restricting the users wishing to access and modify the type management system’s repository contents, *e.g.* adding new types, deleting types and modifying existing types.

Another issue that naming raises is with regard to different type management systems and their interworking or federation. It is likely that in a distributed system several naming schemes for types will be in place, each associated with a given type management system. A client using one naming scheme wishing to access a server in another naming scheme requires some form of guarantee that the type it requires is the one provided. That is, they might have the same name but a guarantee of the semantics is required also. Alternatively, they might have different names but semantically they might be the same. This requires that the naming schemes are scalable and also that they can be made compatible.

Being able to establish that two types are structurally equivalent eases the problem involved in merging (and possibly automating) type naming schemes. That is, being able to state that two types are structurally equivalent might not necessarily imply that they are type equivalent. However, types that are not structurally equivalent cannot be type equivalent. Hence establishing structural equivalence enables a restricted set of type names known to the type management systems to be given, as opposed to the full set of known type names.

In order to check for structural equivalence between types when merging (or federating) type management systems, an efficient relabelling algorithm is essential. As shown above with *P_type* and *R_type*, establishing structural equivalence requires consistent relabelling of the sorts in the signature. In very complex ADTs with numerous sorts many different relabelling possibilities might exist. For two ADTs consisting of n sorts and m sorts respectively, permutation theory gives

$$\frac{n!}{(n-m)!}$$

relabelling permutations. Thus two ADTs which each have five different sorts in their signatures, say, offer 120 relabelling possibilities. Therefore to reduce this complexity several basic, universally known types should be provided. For instance, Booleans, natural numbers and integers represent types which should be universally understood. These may then be removed from the set of sorts contained in the relabelling permutation set.

Other types used in ADTs which are not universally known should be labelled explicitly as locally defined application specific types. To further increase the efficiency of the relabelling algorithm, other time-saving features can be included when checking for structural equivalence. For instance, before relabelling check that the two ADTs under consideration have the same number of sorts in their signature and the same number of operations. If the ADTs do not satisfy these considerations then it can be stated that they are not structurally equivalent, otherwise the relabelling algorithm must be applied to check for structural equivalence.

7.3 The Influence of Causality on Type Checking

The previous section has shown how LOTOS can be used to achieve signature type checking. This approach of type checking based on subsetting of operations has formed the basis of work on type checking of signatures [109, 204]. Unfortunately, this model is inadequate when causality and expected behaviours are introduced into type checking schemes. We investigate this issue through considering the type checking prescribed by ODP for computational interface signatures. We note here that the rules are defined without explicit reference to the causality of the interface.

ODP prescribes the following rules for operational interface types.

Operational interface X is a signature subtype of interface Y if the following conditions hold:

- *for every operation signature in Y, there is an operation signature in X (the corresponding signature in X) which defines an operation with the same name;*
- *for each signature in Y, the corresponding signature in X has the same number and name of arguments;*
- *for each signature in Y, every argument type is a subtype of the corresponding argument type in the corresponding signature in X;*
- *the set of termination names of an operation signature in Y contains the set of termination names of the corresponding signature in X;*
- *for each operation signature in Y, a given termination in the corresponding signature in X has the same number (and names) of result parameters and those parameters associated with X terminations are subtypes of those found in Y terminations.*

These rules may be seen as satisfying a substitutability relation between interfaces. We illustrate this with figure 7.1.

The notation of $a \rightarrow b$ for servers means that the server accepts an input of type a and provides an output of type b. For clients $a \rightarrow b$ means that the client sends an output of type a and receives one of type b. We state that these types are ranges and classical type rules [40] apply to them, *i.e.* a subrange is a subtype.

An interface type relation is used in ODP for two different, but related, reasons: to check whether one interface is an acceptable substitute for another and to bind interfaces. These two areas impose different requirements on types and type checking when causality is considered. Unfortunately, this distinction is not made in ODP and leads to problems in its type model.

Figure 7.1: A Client-Server Type Checking Scenario

7.3.1 Causality and its Effect on Substitution

Checking whether one interface is an acceptable substitute for another should be based on the environment in which the interface is to be used being unaware of any difference in the substitute. Thus causality as an implicit expectation on behaviour is essential. For example, if clients invoke servers then this implies that the servers are passive entities, whereas the clients are the active entities with regard to causing interactions to take place. It might be the case that servers can initiate interactions, *e.g.* through sending notifications to clients. In this situation there is no clear distinction between clients or servers as such. Rather, this is more of a peer-to-peer model. We discuss the problems of confusing peer-to-peer and client-server models shortly in section 7.3.2.

Given that a distinction is made between clients and servers, then in the example given it would be incorrect to substitute server object *A* with client object *C*, even though they support the same operations, since the client expects functions to be performed by other objects. Thus any clients bound to server object *A* initially, would not be serviced by client object *C*. Thus as far as substitutability is concerned, type checking requires like causalities.

Having established like causality, checking substitutability between clients and clients and servers and servers is also different. A client is an acceptable substitute for a second client when the operations it requests are a subset of the operations of the original client. The input parameters associated with the request are subtypes of the original parameters, and all result parameters of the substitute are supertypes of the original parameters. Thus it does not ask for anything the old client never asked for and everything it does ask for is understood by the server. In the diagram, *C* is thus an acceptable substitute for *D* since it has less operations; the operation parameters of *C* are subtypes of the corresponding operations in *D*; and the results of those operations in *C* are supertypes of those in *D*.

A server is an acceptable substitute for another server when it possesses all the operations of the former and possibly more, and any request that could be dealt with by the original server may be dealt with by the substitute. This requires that the input parameters of the substitute server are supertypes of the original server and the return parameters of the substitute server are subtypes of the original server result parameters. In the diagram, *B* is thus an acceptable substitute for *A* since it has more operations; the operation parameters of *A* are subtypes of the corresponding operations in *B*; and the results of those operations in *B* are subtypes of those in *A*.

Thus in effect the type relations between client substitutability and server substitutability are opposite with subsetting and supersetting of types respectively. Hence in our example, *B* is type equivalent (a subtype) to *A* and *C* is type equivalent (a subtype) to *D* based on a syntactic substitutability relationship. As can be seen, ODP only deals with the case of server interfaces and does not make a distinction between the checks required for client and server substitutions.

7.3.2 Causality and its Effect on Binding

With regard to type relationships and binding, if a client-server model is taken where the clients request services to be performed by servers, then opposite causality is required between the interfaces. It would be meaningless to bind two clients (or two servers) since they would both either expect to be served or expect to serve one another respectively. As stated previously, in peer-to-peer systems, *i.e.* systems where servers can initiate interactions such as sending notifications to clients, then these limitations do not exist.

Unfortunately, peer-to-peer systems do not support subtyping between parameters. That is, if a client can invoke an operation on a server and the server can invoke the same operation on the client, then the parameters associated with the operation must be identical for the client and server.

Given that opposite causality has been established, a type relationship for binding requires that:

- the client asks for only a subset of the operations of the server;
- the requests made by the client are understood by the server;
- the responses issued by the server are understood by the client.

In the simplest form of binding (primitive) as given in clause 7.2.3.2 of [109], it is stated that interfaces can be bound provided they have complementary interface signatures. Complementary is defined as identical except for opposite causality. This is too strong a signature compatibility relation since it prohibits the interface of *C* being bound to the interface of *B* above, despite all possible client requests being catered for (understood) by the server and the server responses being understood by the client.

7.4 Type Equivalence in LOTOS

As a result of these considerations, it is apparent that the notion of causality is fundamental to the different aspects that have to be dealt with in establishing type equivalence. We argue that type equivalence should be separated into two areas: type checking for binding and type checking for substitutability. However, as discussed previously, type equivalence based on signatures only is inadequate. We thus propose that type checking should be further separated into three sub-areas. Firstly, syntactic equivalence of signatures should be checked. Secondly, non-functional aspects of types should be checked. Finally, the behavioural aspects of types should be checked. We shall see that syntactic aspects and non-functional aspects are possible to check; behavioural type checking is more problematic however.

To show how these three areas can be dealt with, we take the results of chapters 5 and 6 to show how an extended type checking model can be specified in LOTOS and Z.

7.4.1 An Extended Model of Binding in LOTOS

As shown in section 7.2.5, it is possible to specify interface signature type checking through an approach based on creating the signatures in Act One. When binding between a client and server is done, we require that the operations requested by a client exist in the set of operations that a server offers and that all non-functional aspects that are associated with a client's interface reference are satisfied by the server. The server side of a binding may be represented by:

```
process ServerInterface[ g ...](myRef: IRef, known: IRefSet, ...):noexit:=
 g ?bind: Name !myRef ?pl: HOPList;
 ([ getIRef(pl) IsIn known ] ->
           ....(* already bound to server *)
           ServerInterface[ g ...](myRef,known...)
  []
  [ not(getIRef(pl) IsIn known) and not(getOps(getIRef(pl)) IsSubsetOf getOps(myRef)) ]->
           ...(* operations requested by client not supported by server *)
           ServerInterface[ g ...](myRef,known...)
  []
  [ not(getIRef(pl) IsIn known) and (getOps(getIRef(pl)) IsSubsetOf getOps(myRef))
    and not(getCons(getIRef(pl)) Satisfies getCons(myRef)) ] ->
           ...(* client non-functional aspects not matched by servers *)
    ServerInterface[ g ...](myRef,known...)
  []
  [ not(getIRef(pl) IsIn known) and (getOps(getIRef(pl)) IsSubsetOf getOps(myRef))
    and (getCons(getIRef(pl)) Satisfies getCons(myRef)) ] ->
            ...(* successful behaviour *)
            ServerInterface[ g ...](myRef,Insert(getIRef(pl),known)...)
  []
     ...(* other behaviours restricted to clients in known *)
endproc (* ServerInterface *)
```

Here if the client is already bound to the server then here we refuse the binding request and a recursive call is made. It should be noted that this need not necessarily be the case, *i.e.* the same client might be bound to the same server several times concurrently. Each of these bindings might have different properties associated with them, *e.g.* different sets of operations requested with different constraints. This would require that the server object returned different interface references for each successful binding. For example, instead of inserting the client interface reference into the set *known* for successful binding requests, the server might generate an interface reference which is sent to the client and added to the set *known*. For simplicity, we assume that a successful binding results in the client interface being added to the set of *known* interface references.

If the client is not already bound to the server, *i.e.* not in the set of *known* interface references, then the operations associated with the client's request are checked. If the operations asked for are not available at the server then some error behaviour is taken and a recursive call made. For simplicity here we avoid dealing with the issues involved in type checking the parameters of client and server operations. Rather, we simply state that the operations the client requests should be in the set of operations that the server provides.

In this specification fragment we assume that if the constraints expressly given by the client are not expressly satisfied by the server, *i.e.* by a specific constraint as opposed to its behaviour, then the request to bind is refused. It might also be the case that client constraints can be satisfied by the server without constraints being expressly given. Unfortunately this is not possible to check "on the fly" in LOTOS. As stated in section 7.2.1, LOTOS allows complex behaviours to be specified, but it is not the case that these behaviours can be reasoned about within the specification itself. We discuss this issue in more detail at the end of this chapter.

Finally, if the client asks for legal operations, *i.e.* in the set of operations supported by the server interface reference, and all constraints can be satisfied then some successful behaviour occurs. This might be the sending of a positive response to the client. Following this the client interface reference is then added to the set of known interfaces. Membership of this set then allows access to the other behaviour (not specified here) available at this interface. As stated previously, this other behaviour should realise the set of operations and constraints given by the interface reference *myref*.

This specification fragment uses several Act One operations that should be discussed in more detail. The operation *Satisfies* represents the satisfaction of the client set of non-functional constraints by the server constraints. If a client's constraints are based on those given in the previous chapter, *i.e.* cost, resource usage and timing constraints, then their satisfaction is given by the server:

- having a lower cost than the client is willing to pay;
- having more resources than the client needs;
- producing results at least as fast as the client needs them.

As discussed in section 6.2.1, costs may be calculated once when the interface is accessed or may accumulate as the interface is used. As such, the satisfaction of this relation may involve more than just comparison of two values when binding is done. It might be the case that complex policies are set up where costs decrease as the interface is successively accessed for example. For simplicity here we assume a single value that can be checked once when binding is done.

Resource constraints were specified through action counting in section 6.2.2. In reality, a more complex approach to satisfaction of resource constraints is likely. For example, since numerous clients may be bound to the server, ensuring that the resource constraints given in a binding request are satisfied at all times is likely to require algorithms for dealing with maximum loads and scheduling.

Satisfaction of timing constraints for clients and servers will depend very much on the particular timing requirements themselves. For example, it might be the case that a client has constraints that the results should be returned within some allotted time window. Alternatively, a client might request that results are returned at a specific time. Hence it is not necessarily the case that faster responses are always desireable. Further, these timing constraints might be applied to individual operations or to the interface as a whole. The approach given in section 6.2.3 can be used to deal with such issues, but requires a level of prescriptivity not possible here.

Satisfaction of timing constraints for other interface types, *e.g.* stream interfaces, will almost certainly be much more complex. For example, timing constraints related to the rate, allowed jitter, maximum and minimum delays, etc. will all have to be addressed before any form of successful binding can be achieved.

In the above specification fragment (and in section 6.2.1) we do not have simple parameter lists. Rather, we have a higher order parameter list, where parameters themselves can represent interface references. It is not possible to treat ordinary parameter lists, *i.e. PList*, in the same way as higher order parameter lists, *i.e. HOPList* in LOTOS, as cyclic dependencies would exist between the Act One data types. That is, *Param* is used in *PList*; *PList* is used in *Op*; *Op* is used in *OpSet* and *OpSet* in *IRef*. Thus it is not possible to have an operation in *IRef* like *isParam: IRef→Param*, as this would require *IRef* to be enriched by *Param* which would produce a cyclic dependency.

It is of course quite possible to specify all parameter lists as being higher order, *i.e.* replace all occurrences of *PList* in the process algebra by *HOPList*. We do not do so since in most cases they are not. A higher order parameter list may be represented by:

```
Type HOPList is IRef, PList
     sorts HOPList
     opns makeHPL : -> HOPList
          addParam: Param, HOPList -> HOPList
          remParam: Param, HOPList -> HOPList
          isParam : IRef -> Param
          isIRef  : Param -> Bool
          getIRef : HOPList -> IRef
endtype (* HOPList *)
```

We omit the equations for brevity. They require that a distinction can be made between a parameter representing an interface reference and any other kind of parameter. This might be done for example through checking that a location is associated with the parameter, *i.e. getLoc* as defined in section 5.1.2 does not contain a null value. We note here that the check should not be made on constraints since not all interface references will have constraints associated with them. Further, the check should not be made on an interface having a set of operations as all parameters have a set of operations associated with them.

A server that receives a message from a client wishing to delete its binding may be represented by:

```
process ServerInterface[ g ...](myRef: IRef, known: IRefSet, ...):noexit:=
    g ?deletebind: Name !myRef ?pl: HOPList;
    ( [ not(getIRef(pl) IsIn known) ] ->
           ...(* not bound to server *)
          ServerInterface[ g ...](myRef,known...)
      []
      [ getIRef(pl) IsIn known ] ->
            ServerInterface[ g ...](myRef,Remove(getIRef(pl),known)...)
    []
    (* other behaviours restricted to clients in known *)
endproc (* ServerInterface *)
```

Here the server checks whether the client is bound to that server already. If not then the set of known interface references is unchanged. If it is then the clients interface reference is removed from the set of known interface references.

It should be noted that it might be the case that it is the server that decides to delete the binding. This might be represented as:

```
process ServerInterface[ g ...](myRef: IRef, known: IRefSet, ...):noexit:=
    i; g !deletebind !SomeIRef !pl !Server;
    ServerInterface[ g ...](myRef,Remove(SomeIRef,known)...)
    []
    (* other behaviours restricted to clients in known *)
endproc (* ServerInterface *)
```

Here we do not specify how the interface reference *SomeIRef* was selected to be removed from the set of *known* interface references. The parameter list *pl* may well contain information as to why the binding was terminated.

7.4.2 An Extended Model of Substitution in LOTOS

In the previous section we have seen how a form of type checking between clients and servers can be achieved in the process algebra part of LOTOS. We now consider how it is possible to specify a form of substitution between server interfaces.

As discussed in section 2.2.3, substitution may be regarded as the primary mechanism for addressing evolution. This is the ability to replace a component in a system with a different component and ensure that the system overall is not adversely effected.

In LOTOS, substitution between process definitions directly is not possible in the sense of a new process definition being added to a specification to replace an existing one. The number and configuration of process definitions in a LOTOS specification is static. The behaviour of the specification may, however, result in the instantiation of potentially infinite process instances that allow for different interaction patterns.

It is possible to specify a form of substitution through the redirection of requests that are sent to a server. This is commonly referred to as delegation [165]. Here, the messages sent to an existing process are sent to another process provided the other process has an acceptable behaviour. We have seen above how the syntactic aspects of this behaviour can be checked, *i.e.* through operation subsetting. Further we have seen how certain constraints associated with a given behaviour might be checked. A server that allows for a form of delegation may thus be represented as:

```
process ServerInterface[ g, d ...](myRef: IRef, known: IRefSet, delRef: IRef ...):noexit:=
     g ?op: Name !myRef ?pl: HOPList;
     ( [ (op eq delegate) and (makeOp(op,pl) IsIn getOps(myRef)) and
          (getCons(getIref(pl)) Satisfies getCons(myRef)) ] ->
          ServerInterface[ g ...](myRef,known,getIRef(pl)...)
    []
       (* behaviour for unsuccessful delegation operation *)
    []
    ...
       (* if not delegation operation and delRef eq NULL *)
       (* check parameters and have behaviour as before *)
    ...
    []
       (* if non-delegation operation with ok details and delRef ne NULL *)
             d !op !delRef !pl;
             d ?res: Name !myRef ?pl : HOPList;
             g !res !getIRef(pl) !pl;
             ServerInterface[ g, d ...](myRef,known,delRef...)
endproc (* ServerInterface *)
```

Here we include an extra parameter in the formal parameter list *delRef* which is a reference to a process that process *ServerInterface* delegates requests to. When process *ServerInterface* is instantiated, this will be set to *NULL*. The server process behaviour then checks that this formal parameter is not set to any value other than *NULL* when requests are received. For simplicity, we do not deal with the situation where we have multiple interfaces that the behaviour is delegated to, although this would be quite possible to specify.

Should a process synchronise at gate *g* with the operation name *delegate* then various checks are done. Firstly, the process checks that the details of the request are valid, *i.e.* that delegation is a permitted operation for this interface. Following this, it checks that the constraints associated with the delegation offer satisfy the existing interface reference constraints. It should be noted that if this is the case then the constraints associated with the *known* set of interface references should also be satisfied.

If the operation invoked at gate *g* is not a delegation operation, then the behaviour depends on whether the process has already delegated responsibility, *i.e.* if *delRef* is not *NULL*. All guards that limit access to the process behaviour include a check on whether this interface reference is *NULL* or not. If it is *NULL* then behaviour being guarded should be that behaviour required to satisfy the interface reference. If it is not *NULL* and the details of the request are satisfactory, *i.e.* the requested operation is permitted, then the details of the request are forwarded at gate *d* to the delegated process. Responses are returned which are then passed on to the original requestor.

7.5 Type Equivalence in Z

We have seen in chapter 5 how Z could be used to specify aspects of computational interfaces consisting of collections of syntactic units (operation, signal and flow signatures) and how these units could be related directly to behaviours. In chapter 6 we presented different constraints that might influence behaviours. Specifically, we focussed on cost, time and resource usage constraints. In this section we shall see how certain of these specification fragments can be applied to develop a more complete model of type checking in Z. A similar approach to develop a more complete type checking model for distributed systems may be found in [179].

ODP computational interfaces consist of interface signatures, behaviour specifications and environment contracts. Environment contracts may be regarded as agreements on the behaviour offered by the interface and expected from the environment for the interface to function correctly in the environment[5]. Environment contracts may contain constraints on quality of service, *e.g.* throughputs and delays, as well as usage and management constraints. These might include particular location constraints on the interface usage as well as the need for selected distribution transparencies (see section 2.1.3).

ODP prescribes rules that have to be satisfied when composing systems or replacing one system by another. These rules are based on interface signatures only. No account is taken of behavioural issues or consideration of environment contracts. To a large extent, this is only natural since ODP as a generic architecture cannot prescribe explicit behaviour or environment contract contents. Nevertheless we shall see how it is possible to specify satisfaction of behavioural and environment contract aspects in a generic manner in Z. For simplicity we focus on a subset of the non-functional constraints that might be associated with interfaces, namely delays and throughputs.

We note here that we do not consider all aspects of all interfaces, *i.e.* operational, signal and stream interfaces. Rather we show how syntactic and behavioural aspects of operational interfaces can be checked, and how non-functional aspects of stream interfaces can be checked.

7.5.1 Checking Interface Signatures when Binding in Z

Two operational interfaces are syntactically equivalent and can be composed (bound) when they have opposite causalities and they do not issue requests or responses that are not understood. To specify this, it is necessary to introduce a basic idea of subtyping between parameters. For simplicity we assume that this relation is based on sequences of parameters. The extension from relationships between types to relationships between sequences of types is given in [40]. We may represent this relationship as:

$$isSubtype : PList \leftrightarrow PList$$

Two interfaces with opposite causality can be bound successfully with regard to messages being understood when the invocation names of the client exist in the server and the parameters associated with the request are subtypes of those in the server interface. In addition, the termination names of the server must exist in the client and the parameters associated with the response are supertypes of those in the server interface. This can be represented by:

$$\begin{array}{l} BindSyntaxOk : OpIntSig \leftrightarrow OpIntSig \\ \hline \\ \forall ois_c; ois_s : OpIntSig \mid (ois_c; ois_s) \in BindSyntaxOk \bullet \\ \quad ois_c.role = Client \land ois_s.role = Server \Rightarrow \\ \quad (\forall int_c : IntSig \mid int_c \in ois_c.ints \bullet (\exists int_s : IntSig \bullet int_s \in ois_s.ints \land \\ \qquad int_c.inv.invName = int_s.inv.invName \land (int_c.inv.inArgs; int_s.inv.inArgs) \in isSubtype \land \\ \qquad (\forall tt_s : TermTemplate \mid tt_s \in int_s.terms \bullet (\exists tt_c : TermTemplate \bullet tt_c \in int_c.terms \land \\ \qquad\quad tt_s.termName = tt_c.termName \land (tt_s.outArgs; tt_c.outArgs) \in isSubtype)))) \end{array}$$

This relation provides the basis for syntactic interworking between clients and servers, *i.e.* it provides the minimum requirements necessary to start dealing with issues such as openness. For brevity, we do not make this a symmetric relationship. For similar reasons we also do not produce the Z text representing the syntactic compatibility of stream and signal interfaces. These specifications may be found in [178].

[5] and in turn for the environment to expect the correct behaviour from the interface.

7.5.2 Checking Non-Functional Aspects when Binding in Z

Chapter 6 introduced a collection of non-functional aspects that might be associated with a given interface or system. As stated previously, constraints on location, usage and management may also be given. These constraints may simply be represented as given types. For example, transparency constraints may be modelled as:

$$Transparency ::= Access \mid Location \mid Replication \mid :::$$

We may collect all of the constraints that might apply to a given interface or system into a parameterised free type definition. We name all of these constraints of systems collectively as *Constraints*. These can be represented as:

$$\begin{aligned} Constraints ::= & \; maxCostInterface\langle CostConsInterface\rangle \mid \\ & maxResInterface\langle SetInterfaceResourceLimit\rangle \mid \\ & maxD\langle maxDelay\rangle \mid \\ & maxTP\langle maxThruPut\rangle \mid \\ & requiredTransparencies\langle Transparency\rangle \mid :::\end{aligned}$$

Here the dots imply that this set of assertions on system is not complete. With this model, we may represent an environment contract as:

$$\begin{array}{|l} EnvCon \\ \hline needs : \mathbb{F}\, Constraints \\ promises : \mathbb{F}\, Constraints \\ \hline \end{array}$$

We note here that the environment contract is divided into two groups: constraints on the expected behaviour of the environment and constraints on the behaviour of the interface with which the environment contract is associated. It should be pointed out that these are finite sets of constraints and not non-empty finite sets of constraints. It might well be the case that there are no constraints associated with an interface or system. In this case the signature and behaviour are all that exist. Where explicit constraints are given, several conditions on the interfaces are imposed when binding is to take place. Firstly, the assertions should be understandable. Thus as with interface signature binding, the constraints will have a syntactic form that will be understood between systems. Secondly, the constraints should ideally be easily checked. In most cases a constraint corresponds to a name and value pair. More complex constraints can exist as well as constraints that are dependent on one another, *e.g.* more cost allows a higher throughput to be achieved. It is possible to specify such relationships between constraints, but for simplicity we focus only on independent constraints represented by names and values. Satisfaction of constraints will depend upon these values.

A consumer interface may be bound to a producer interface when:

- The maximum throughput the consumer can consume is greater than the maximum throughput the producer can produce.
- The maximum delay a consumer can tolerate is less than the maximum delay a producer exhibits [6].

The intuitive meaning here is that the consumer can consume at least as fast as the producer can produce. Further, the producer delays between frame production are within the range of acceptable delays of the consumer. The satisfaction of constraints for throughputs and delays given here can be represented as:

$$\begin{array}{|l} Satisfies : Constraints \leftrightarrow Constraints \\ \hline \forall mft_1; mft_2 : mmFlowType;\ c_1; c_2 : Constraints \mid (c_1; c_2) \in Satisfies \bullet \\ \quad (c_1 = maxTP(mft_1; maxThruPut(mft_1)) \land c_2 = maxTP(mft_2; maxThruPut(mft_2)) \land \\ \qquad maxThruPut(mft_1) \geq maxThruPut(mft_2)) \lor \\ \quad (c_1 = maxD(mft_1; maxDelay(mft_1)) \land c_2 = maxD(mft_2; maxDelay(mft_2)) \land \\ \qquad maxDelay(mft_1) \leq maxDelay(mft_2)) \end{array}$$

[6] For simplicity we ignore issues related to network latency, although we acknowledge that this is a considerable simplification.

This states that if the constraint of interest is throughput, then the constraint of one information flow is satisfied by another with a higher throughput. If the constraint is delay then one information flow satisfies another when it has less delays between frames. We shall show in section 7.5.7 how this satisfaction relation can be attached explicitly to interfaces to check whether they can interwork successfully. A similar approach to satisfaction of environment contracts was presented in [177].

7.5.3 Checking Interface Behaviours when Binding in Z

Establishing whether clients can be bound to servers and have some desirable behaviour *a priori* is in general not possible to compute for anything other than simplistic behaviours. Nevertheless, it is possible to specify certain desirable features that one would like bound interfaces to possess. For example, one feature might be that the client and server do not have contradictory behaviours.

We note here that the basis for interaction is that the clients and servers have compatible signatures, *i.e.* their associated signatures are in the relation *BindSyntaxOk* given previously. Clients and servers having non-contradictory behaviours may be represented as:

$$NonContraBehOk : OpIntTemplate \leftrightarrow OpIntTemplate$$

$$\forall ci; si : OpIntTemplate \mid (ci; si) \in NonContraBehOk \bullet (ci{:}ops; si{:}ops) \in BindSyntaxOk \land (\exists res : behspec \bullet res = (si{:}sbs \cup ci{:}cbs)^+)$$

This states that the transitive closure of the client and server behaviours should exist. This is only true if the client and server do not wish to perform contradictory actions, *i.e.* the client wishes to perform action *x* followed by *y* and the server wishes to perform *y* followed by *x* say. This specification fragment requires that all client and server actions not observable in the interface, *i.e.* internal actions or actions at other interfaces, are distinct. To overcome this restriction it is possible to modify the relation so that contradictions do not exist in the set of actions that they have in common. For brevity, we omit the Z text. The same idea of checking that the transitive closure of the (restricted) relation exists however.

We note that this relation also requires that the client and server behaviours themselves are irreflexive. This may not be the case, especially where concurrent processing is possible, *e.g.* where the behaviour contains actions such as $a_{|||}b$ which can lead to the actions (*a*,*b*) and (*b*,*a*) being in the relation.

Whilst having non-contradictory behaviours might be a useful guide to checking that interfaces do not have some "bad" behaviour, *e.g.* deadlock, non-contradictory behaviour does not mean to say that the interfaces can interact successfully and have some "good" behaviour. Without being specific regarding the modelling of behaviours, it is not possible to prescribe what constitutes "good" behaviour. One useful thing to specify though is that once a server receives an invocation from a client, it does not have a sequence of states that does not lead to an associated termination being sent.

$$BindBehOk : OpIntTemplate \leftrightarrow OpIntTemplate$$

$$\begin{aligned}&\forall si; ci : OpIntTemplate \mid (ci; si) \in BindBehOk \bullet \\ &\quad (ci; si) \in NonContraBehOk \land \\ &\quad (\forall is : IntSig;\ it : InvTemplate;\ tt : TermTemplate;\ s_1; s_2; s_3; s_4; s_5; s_6; s_7; s_8 : State \mid \\ &\qquad is \in ci{:}ops{:}ints \land it = is{:}inv \land tt \in is{:}terms \land \\ &\qquad (isInvAct(it; s_1; s_2); isTermAct(tt; s_3; s_4)) \in ci{:}opbs^+ \land \\ &\qquad (isInvAct(it; s_5; s_6); isTermAct(tt; s_7; s_8)) \in (si{:}opbs \lhd isInvAct(it; s_5; s_6))^+ \land \\ &\qquad isInvAct(it; s_1; s_2) = isInvAct(it; s_5; s_6) \land isTermAct(tt; s_3; s_4) = isTermAct(tt; s_7; s_8) \bullet \\ &\qquad\quad (\forall ss_1; ss_2 : seq State \mid ss_1 \frown \langle s_5; s_6\rangle \frown ss_2 \frown \langle s_7; s_8\rangle \text{ prefix } si{:}ophis \bullet \\ &\qquad\quad (\forall ss_3 : seq State \mid ss_1 \frown \langle s_5; s_6\rangle \frown ss_3 \text{ prefix } si{:}ophis \bullet ss_3 \text{ prefix } ss_2 \lor \langle s_5; s_6\rangle \text{ in } ss_3)))\end{aligned}$$

Several things should be noted here. Firstly, we state that the behaviours of the client and server do not contradict one another. Secondly, we state that the invocation from the client to the server represents the same action, as does the termination from the server to the client. Thus for interactions to take place, the actions must be the same. It might be considered that this goes against the subtyping possibilities for parameters. We argue that when an interaction takes place, the subtyping rules either are satisfied, *e.g.*

coercion or some other mechanism is used to convert the type to one expected, or no interaction takes place. Since we only deal with successful binding here the erroneous case is not considered. The final two universal quantification predicates state that all sequences of server states following a particular invocation lead to the associated termination, *i.e.* sequences that do not lead to the termination are prohibited.

We note here that we focus only on the behaviour specification consisting of those actions following the invocation being considered, *i.e.* we avoid the problem of terminations associated with previous invocations. For simplicity we assume that the invocation does not occur again before a termination has been issued.

7.5.4 Checking Interface Signatures when Substituting in Z

A server interface is an acceptable substitute for another server interface when it is syntactically compatible, *i.e.* it extends the signature. This can be represented by:

$$
\begin{array}{l}
SubstituteSyntaxOk : OpIntSig \leftrightarrow OpIntSig \\
\hline
\forall ois_{sub}; ois : OpIntSig \mid (ois_{sub}; ois) \in SubstituteSyntaxOk \bullet \\
\quad ois{:}role = Server \land ois_{sub}{:}role = Server \Rightarrow \\
\quad\quad (\forall int : IntSig \mid int \in ois{:}ints \bullet (\exists int_{sub} : IntSig \bullet int_{sub} \in ois_{sub}{:}ints \land \\
\quad\quad\quad int_{sub}{:}inv{:}invName = int{:}inv{:}invName \land (int{:}inv{:}inArgs; int_{sub}{:}inv{:}inArgs) \in isSubtype \land \\
\quad\quad\quad (\forall tt_{sub} : TermTemplate \mid tt_{sub} \in int_{sub}{:}terms \bullet (\exists tt : TermTemplate \bullet tt \in int{:}terms \land \\
\quad\quad\quad\quad tt_{sub}{:}termName = tt{:}termName \land (tt_{sub}{:}outArgs; tt{:}outArgs) \in isSubtype))))
\end{array}
$$

This relation provides the basis for syntactic substitution between servers, *i.e.* it provides the minimum requirements necessary to start dealing with issues such as evolution. For brevity, we do not give here the conditions for client substitutions. These are similar but with the roles set to client and the subtyping of parameter rules for invocations and terminations reversed.

7.5.5 Checking Non-Functional Aspects when Substituting in Z

As previously, we consider assertions on the maximum throughput and delays of multimedia flows. One consumer interface may be substituted for a second consumer interface when:

- The maximum throughput the substitute consumer can consume is greater than the original consumer.
- The maximum delay the substitute consumer can tolerate is greater than the original consumer.

A producer interface may be substituted for another producer interface when:

- The maximum throughput the substitute producer can produce is greater than the original producer.
- The maximum delay between frames of the substitute producer is less than that of the original producer.

We note here that these constraints do not imply that the producer must produce faster. They only represent the fact that the producer is able to produce faster. For example, if a consumer wants a producer to work at its maximum rate, then a substitute producer should be able to achieve this rate also. For brevity, we do not produce the Z texts showing satisfaction of constraints when substituting for producers or consumers. These are similar to those given previously for constraint satisfaction with the predicates modified accordingly.

7.5.6 Checking Interface Behaviours when Substituting in Z

Once signature compatibility has been checked we need to check behavioural compatibility. Behavioural compatibility for server substitutes requires that, whatever sequence of states the original server could produce between an invocation and its associated termination, then the substitute server cannot have a sequence of states that does not lead to these sequences. A basis for establishing this is that they are

syntactically compatible and that the behaviour of the original server is contained within the behaviour of the replacement. This can be represented as:

$$SubContainsBehOk : OpIntTemplate \leftrightarrow OpIntTemplate$$

$$\forall si_1; si_2 : OpIntTemplate \mid (si_1; si_2) \in SubContainsBehOk \bullet (si_1.ops; si_2.ops) \in SubstituteSyntaxOk \land si_2.opbs \subseteq si_1.opbs$$

Ensuring that all invocations leading to terminations in the original server are exhibited by the replacement server may be represented as:

$$SubstituteBehOk : OpIntTemplate \leftrightarrow OpIntTemplate$$

$$\begin{aligned}
&\forall si_1; si_2 : OpIntTemplate \mid (si_1; si_2) \in SubstituteBehOk \bullet \\
&\quad (si_1; si_2) \in SubContainsBehOk \land \\
&\quad (\forall is : IntSig;\ it : InvTemplate;\ tt : TermTemplate;\ s_1; s_2; s_3; s_4; s_5; s_6; s_7; s_8 : State \mid \\
&\qquad is \in si_1.ops.ints \cap si_2.ois.ints \land it = is.inv \land tt \in is.terms \land \\
&\qquad (Inv(it; s_1; s_2); Term(tt; s_3; s_4)) \in (si_2.opbs \Rightarrow Inv(it; s_1; s_2))^+ \land \\
&\qquad (Inv(it; s_5; s_6); Term(tt; s_7; s_8)) \in (si_1.opbs \Rightarrow Inv(it; s_5; s_6))^+ \land \\
&\qquad Inv(it; s_1; s_2) = Inv(it; s_5; s_6) \land Term(tt; s_3; s_4) = Term(tt; s_7; s_8) \bullet \\
&\qquad (\forall ss_1; ss_2 : \text{seq } State \mid ss_1 \frown \langle s_5; s_6 \rangle \frown ss_2 \frown \langle s_7; s_8 \rangle \text{ prefix } si_1.ophis \bullet \\
&\qquad\quad (\forall ss_3 : \text{seq } State \mid ss_1 \langle s_5; s_6 \rangle \frown ss_3 \text{ prefix } si_1.ophis \bullet ss_3 \text{ prefix } ss_2 \lor \langle s_5; s_6 \rangle \text{ in } ss_3)))
\end{aligned}$$

Several points are worth mentioning about this specification fragment. Firstly, we state that the invocations and terminations in the two servers are the same. This allows us to relate the states of the two servers directly. Thus s_1 and s_5 for example might be completely different, but provided the invocation *it* can occur in these two states then we do not need to know any more about them. Similar considerations apply to the other states given. This lack of awareness is deliberate. Other work has focused in great detail on state relationships and their relationship to subtyping. Cusack for example [54] has developed a detailed theory for acceptable substitute operations based on state comparisons in a Z extension [55]. Our treatment of states here is largely as labelled placeholders for reasoning about action orderings. This approach is more in the style of process algebras. We argue that this is more directly relevant to us here since it can be applied directly within the specification as a basis for reasoning. Cusack's approach requires considerations about subtyping issues to be handled outside the specifier's domain. That is, such issues are normally considered when reasoning about the specification. Here we want to specify what the issues are directly when composing systems or substituting them for one another. For substitutability, we would argue that reasoning about behavioural subtyping can largely be avoided if one assumes that invocations and terminations are the same. This is not an overly restrictive assumption given that one would expect the replacement to have exactly the same behaviours where the behaviours overlap[7].

The final two universal quantification predicates state that for all invocations and subsequent terminations associated with a given server, a substitute server cannot have a sequence of states that does not lead to the same termination once the given invocation has occurred. For simplicity, we assume that the state sequences leading up to the state in which the invocation is possible are the same.

7.5.7 Extended Type Checking Model in Z

Computational interface templates consisting of signatures, behaviour specifications and environment contracts may be represented as:

$$\begin{aligned}
&ExtendedOpIntTemplate \\
&opint : OpIntTemplate \\
&ec : EnvCon
\end{aligned}$$

To specify truly open systems requires that all syntactic, behavioural and non-functional considerations contained within an (extended) operational interface template are satisfied when binding takes place. Thus

[7] Of course, this assumes that the original server functioned correctly in the first place and was not being substituted due to erroneous behaviour.

all constraints put forward in the environment contract should be satisfied by the interface behaviour (if they were *promises*) or satisfied by the environment (if they were *needs*). This can be represented as:

$$
\begin{array}{l}
OpenComposition : ExtendedOpIntTemplate \leftrightarrow ExtendedOpIntTemplate \\
\hline
\forall C; S : ExtendedOpIntTemplate \mid (C; S) \in OpenComposition \bullet \\
\quad (C{:}opint; S{:}opint) \in BindBehOk \wedge \\
\quad (\forall c_1; c_2 : Constraints \mid c_1 \in S{:}ec{:}needs \wedge c_2 \in C{:}ec{:}needs \bullet \\
\qquad (\exists c_3; c_4 : Constraints \mid c_3 \in S{:}ec{:}promises \wedge c_4 \in C{:}ec{:}promises \bullet \\
\qquad\quad (c_4; c_1) \in Satisfies \wedge (c_3; c_2) \in Satisfies))
\end{array}
$$

To specify truly evolving systems in Z require that all syntactic and behavioural considerations are upheld by systems. Further, all explicit statements the system promises to uphold are upheld by the substitute system and the constraints the replacement system places on the environment are satisfied by the constraints of the older system. This can be represented as:

$$
\begin{array}{l}
EvolvingSystem : ExtendedOpIntTemplate \leftrightarrow ExtendedOpIntTemplate \\
\hline
\forall S_1; S_2 : ExtendedOpIntTemplate \mid (S_1; S_2) \in EvolvingSystem \bullet \\
\quad (S_1{:}bsi; S{:}bsi) \in SubstituteBehOk \wedge \\
\quad (\forall c_1; c_2 : Constraints \mid c_1 \in S_2{:}ec{:}needs \wedge c_2 \in S_2{:}ec{:}promises \bullet \\
\qquad (\exists c_3; c_4 : Constraints \mid c_3 \in S_1{:}ec{:}needs \wedge c_4 \in S_1{:}ec{:}promises \bullet \\
\qquad\quad (c_1; c_3) \in Satisfies \wedge (c_4; c_2) \in Satisfies))
\end{array}
$$

7.6 Summary

This chapter has shown how types are critical to distributed systems design and development. An approach to specify types in a form that allows for their structural similarity to be checked was presented in Act One. A complete specification showing how type systems can be specified and checked so that type relationships can be established, *e.g.* subtyping, is presented in Appendix A. This specification also allows assertions on type relationships to be made and checks performed on the possible validity of those assertions.

We have seen how LOTOS can be used to model interface references consisting of sets of operations with associated constraints. This approach enabled a form of type checking to be achieved with LOTOS. This type checking was applied to decide whether one interface wishes to accept a binding request from another interface based on the operations and constraints of the other interface. In addition we have shown how substitution of systems could be achieved abstractly through a form of delegation based on interface reference contents.

One issue with this work in general is that we have largely avoided dealing with behavioural aspects of interfaces. The interface references we model and check are essentally syntactic structures withe constraints attached. Ideally, we would like to be able to check behavioural aspects of interfaces also. For example, the model of interface references presented here does not allow checks that the sets of operations and constraints are really exhibited by the process with which they are attached. Such checks would certainly be required if one system were to be replaced by another. Unfortunately, the checking of such behaviours is not possible to achieve within LOTOS itself. Behavioural issues regarding the specification are normally considered when the specification is tested for example. Testing for bisimulations “on the fly” for example is currently an active area of research [79]. Certain tools [69] can indeed check for such relationships, however, they place overly restrictive demands on the specification structure, *e.g.* typically recursion is not permitted. Such restrictions whilst making the checks decidable, do not allow realistic systems to be specified, *e.g.* objects that can continually accept requests say.

Despite this, we have seen that it is possible to specify powerful relations in Z representing desirable features of systems. For example, specifying that servers always return terminations when invoked by a client. We have seen also that Z is well suited to modelling constraints on behaviours. As discussed in section 6.4, it could be argued that one problem with Z is that it is very good at making assertions on behaviours, but limited in the features it offers for verifying such assertions. We shall see in the following chapter how certain assertions can be checked when we specify a producer and consumer flow configuration.

Chapter 8

Applying the Architectural Semantics

In this chapter we show how the architectural semantics developed in the previous chapters can be applied to two case studies. The case studies selected are a trading system and a producer and consumer flow configuration. We specify the trader system using the architectural semantics developed for LOTOS and the producer and consumer flow configuration using the architectural semantics developed for Z.

8.1 Specifying a Trading System in LOTOS

ODP aims to provide distribution-transparent utilisation of services over heterogeneous environments. In order to use services, users need to be aware of potential service providers and to be capable of accessing them. Since sites and applications in distributed systems are likely to change frequently, it is advantageous to allow late binding between service users and providers. If this is to be supported, a component must be able to find appropriate service providers dynamically. The ODP trading function provides this dynamic selection of service providers at run time. The interactions that are necessary to achieve this are shown in figure 8.1.

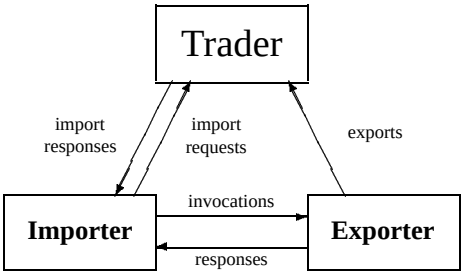


Figure 8.1: A Trader and its Users

Here a trader accepts a *service offer* from an *exporter* wishing to advertise its services. A service offer contains the characteristics of a service that a service provider is willing to provide. We note here that the service provider need not necessarily be the exporter. Similarly the service user may not be the importer. The trader then stores these service offers for use by importers.

A trader accepts service requests from importers of services. These represent requirements on available services that a trader may or may not have access to. Upon receipt of a request from an importer, the trader searches its store of service offers to see if any offers match the importer's service request. If any matching offers are found they are returned to the importer, which may then interact directly with the service.

It should be pointed out that a trader might not itself have a service offer that matches an importer's request. In this case, a trader can check whether any other traders it 'knows' might satisfy the import request. This is known as federated trading. The establishment of trading links between traders is not addressed here, although an architectural approach can be applied to develop a trading federation.

The trader standard [115] identifies numerous operations that a trader should support. In this section we consider only a subset of these. Specifically we deal with exporting a service offer and importing a service offer. Both of these operations have certain parameters that have been prescribed in the trader standard. We consider these in turn.

8.1.1 Parameters Associated with the Export Operation

The export operation allows trader users to advertise services. As such the associated parameters should provide enough information for a trader to establish whether this service is the one that a particular importer wants. The parameters associated with an export operation include:

- an identifier for the exporter of the service offer;
- the type of the service being exported;
- the values of the service properties associated with this type of service;
- the current values of the service offer properties associated with this service offer;
- a reference to an interface at which the current values of the service offer properties can be established;
- an interface reference that can be used to access the service.

The first bullet point here is used by the trader to decide whether that exporter is allowed to export the service offer, *i.e.* whether they have already registered with the trader. The type of service being offered is given by the set of operations that the service supports and a set of service properties that are associated with that service. Service properties and service offer properties represent extra information that the signature of the interface alone does not capture (*i.e.* our model of constraints given in chapters 5 and 6). Service properties may loosely be regarded as static information associated with the service. Service offer properties represent more dynamic information that might be associated with an interface. For example, if the service was a printer service then a static property might be cost and a dynamic property might be current usage.

The current values associated with the service properties and service offer properties are included in the export operation. Given that the properties themselves might change, an interface reference is given to a service offer evaluator interface. This allows the current values of the service properties and service offer properties to be checked. The OMG equivalent of this is the property service [89].

Finally the export offer contains an interface reference that can be used by the importer to access the service. For simplicity here we assume that this interface reference is compatible with the service type, service properties and service offer properties associated with the export operation. It might well be the case that the trader (or a type management system acting on the trader's behalf) performs some checks to ensure that this is the case.

The result of an export operation is either an appropriate error message or an identifier for the service that has been exported to the trader. This identifier can then be used for example to remove (withdraw) the exported service from the set of services that the trader has access to.

8.1.2 Parameters Associated with the Import Operation

The importer operation allows trader users to find services that meet their particular criteria. As such the parameters associated with the import operation should provide enough information that the trader can search through the service offers that have been exported and return only those offers that meet the importer's criteria. The parameters associated with the import operation include:

- an identifier for the importer making the import request;
- the type of the service to be imported;
- the service properties that the importer is interested in;
- the service offer properties that the importer is interested in;
- matching constraints;
- selection preferences;
- an importing policy.

As with the export operation, the trader requires that importers are identified to ensure that users are registered with the trader. The import operation must identify the type of the service that is to be imported, *i.e.* the set of operations that the importer is interested in and service properties and offer properties associated with this service. The importer may also enumerate the service properties and offer properties whose values they want returned along with matching offers themselves.

The matching constraints are used to determine what constitutes a good enough match. For example, the matching constraints might state that only interfaces costing less than a certain amount should be returned, *i.e.* the service property associated with the service representing cost should be below a certain amount. The selection preference can be used to order these returned offers in some way, *e.g.* cheapest first. Finally, the importer's policy can be used to restrict the scope of the search, *e.g.* only search the local trader (and not remote ones) or stop the search after a certain amount of time.

The result of an import operation is either an appropriate error message, or one or more (possibly ordered) interface references. These interface references can subsequently be used for accessing the services.

8.1.3 Structuring a Trader with Importers and Exporters

In order to trade services, all importers and exporters must be aware of the trader's existence and be able to interact with it. It is quite possible here to specify only the trader itself and not deal with the issues involved in the subsequent binding that might or might not occur between importers and exporters. Instead, we show how LOTOS can be used to specify such dynamic communications, *i.e.* where importers get references to services they did not know about previously and interact with them "on the fly".

In order to specify such dynamic systems it is necessary to impose constraints on the structure of the specification. As discussed in section 4.2.1, all communication takes place in LOTOS through gates. Gates are assigned statically to processes and they may not be passed around between processes. As a result, to model systems where dynamic communication paths may be established requires that all processes share at least some common gates. For simplicity we assume that all processes are aware of the trader. An outline of a specification structure showing how importers and exporters can be structured with a trader might be:

```
trader[ t ]
|[t]|
(importers[ t,g, .... ] |[ g,.. ]| exporters[ t,g, ....])
```

Here all importers and exporters should (potentially) be able to interact with the trader. We use specific gates for interacting with the trader. It might also be the case that the trader has more than one gate associated with it, *e.g.* different gates for importers and exporters might be used. We consider a trading system where all communication with the trader is through a single gate. Importers and exporters that are likely to interact with one another at some future time[1] should share one (or more) gates that can be used for binding using the information returned from the trader, *i.e.* the returned interface references.

This structure provides communication channels between importers, exporters and the trader, but it does not allow any form of checking to be done inside the specification on the behaviours that may or may not take place. We have seen in chapter 5 that it is possible to model interface references in LOTOS as first class citizens. To allow importers and exporters to trade successfully with the trader and subsequently bind to one another we need to consider the interface reference for the trader.

[1] It might be the case that all importers and exporters have the potential to bind with one another, in which case they should all share a common gate.

8.1.4 Specifying a Trader Interface Reference

As given in chapter 5, an interface reference may be represented by an Act One data type consisting of a set of operations, a location and a set of constraints. For simplicity here we assume that the trader interface reference does not have any constraints attached to it. In reality, a trader might for example charge for services or have other constraints that might equally apply, *e.g.* the maximum number of requests that can be processed at any one time.

As discussed in sections 4.2.8 and 4.2.9, formal specification is concerned with the building of models of systems and using these models to reason about properties of those systems. Given this level of abstraction, it is not the case that formal languages deal with real-world locations in space or time. Nevertheless we argued in sections 4.2.8 and 4.2.9 that LOTOS could be used to model and reason about location in an abstract manner. For brevity we introduce a simple model of location in Act One. This might be represented as:

```
type Location is
     sorts Loc
     opns locX, locY, locZ: -> Loc
     eqns ...
endtype (* Location *)
```

We omit the equations for brevity. We consider in more detail how issues related to location can be used to influence trading generally at the end of this chapter.

An interface reference for a trader providing only import and export operations can be constructed by the following LOTOS fragment:

```
let tref: IRef = makeIRef(locZ,insert(import,insert(export,{})),{}) in
```

For simplicity here we assume the existence of the import and export operations. We state that the trader interface reference is located at location *locZ*, it supports the operations of import and export and there are no constraints attached to it. All importers and exporters that wish to import from or export to this trader should have access to this interface reference[2]. This can be represented as:

```
let tref: IRef = makeIRef(locZ,insert(import,(insert(export,{}))),{})
in
    trader[ t ](tref...)
    |[t]|
    (importers[ t,g, .... ](...tref...) |[ g,.. ]| exporters[ t,g, ....](...tref...))
```

As discussed in section 5.1.2, interface references modelled as Act One data types should be realised by the behaviour of the process with which they are associated. For a trader this implies that it supports the operations of import and export. An initial structure for a trader specification providing import and export operations might be:

```
process trader[ t ](tref: IRef, ...):noexit:=
     t ?op: Name !tref ?pl: HOPList;
     ( [ not(makeOp(op,pl) IsIn getOps(tref)) ] ->  ... (* return error *)
       []
       [ makeOp(op,pl) eq export ] ->               ... (* process request *)
       []
       [ makeOp(op,pl) eq import ] ->               ... (* process request *)
       |||
       trader[ t ](tref ...))
endproc (* trader *)
```

We expand upon this structure shortly after we have considered how exporters and importers might be structured.

[2] And a communication channel (gate) should exist for interacting with the trader.

8.1.5 Specifying the Structure of Exporters

To show how LOTOS can be used to model dynamic systems, we consider how services can be specified. To be less abstract we deal with services consisting of specific operations and properties. Consider an exporter offering the operations $X_1; X_2; X_3$ at location *locX*, with service properties based on cost and with service offer properties based on the expiry date of the service offer. If the cost for using this service, *i.e.* invoking any of the operations provided, was set at *costX* and the service offer expiry date was set at *timeX*, then the exporter interface reference might be represented as:

```
let eref: IRef = makeIRef(locX,
                          insert(X1, insert(X2, insert(X3,{}))),
                          makeCons(insert(costX,{}),{},insert(timeX,{})))
in
```

Here we use the model of constraints given previously, *i.e.* based on cost, resource usage and time constraints. We note here that no constraints were given for resource usage, *i.e.* the second argument of the *makeCons* operation is an empty set.

Using this model of exporter interface references and the trader interface reference given previously, an exporter may be represented as:

```
process Exporter[ g,t...](eid: ID, eref, tref: IRef...):noexit:=
  t !expName !tref !addParam(eid,
                   addParam(getOps(eref),
                   addParam(getCC(getCons(eref)),
                   addParam(getTC(getCons(eref)),
                   addParam(isParam(eref),
                   addParam(isParam(eref),makeHPL))))));
       (t !expName !eref ?res: Message;
        Exporter[ g,t...](eid,eref,tref,...)
        []
        t !expName !eref ?id: ID;
        Exporter[ g,t...](eid,eref,tref,...))
  []
  ...
  g ... (* offering operations X1, X2, X3 *)
  ...
  []
  g ... (* offering operations to return the current service properties *)
endproc (* Exporter *)
```

Several things should be noted here. Firstly, we assume that there is some way in which the exporter can be identified (*eid*). We do not consider in detail how this is done here but note that it is possible to achieve, *e.g.* through restricting instantiations of object templates to ensure that new identities are guaranteed [44, 147, 110]. We note also that we do not distinguish between the interface reference and the interface reference for the service offer evaluator interface. This implies that if the values of dynamic properties associated with the service are to be established, then this is achieved through interacting with the service directly as opposed to some other object.

8.1.6 Specifying the Structure of Importers

We are not interested in how importer interface offers are represented here. We only require that they exist and can be used for communicating with the trader.

An importer may be structured as follows:

```
process Importer[ g,t...](iid: ID, iref, tref: IRef, known: IRefSet...):noexit:=
  t !impName !tref !addParam(iid,
                   addParam(insert(X1, insert(X3, {})),
                   addParam(makeCons({}, insert(resAvailable, {}), {}),
                   addParam(makeCons({}, {}, insert(expiryDate, {})),
                   addParam(insert(maxCost,{}), {}, insert(minDate,{}),
                   addParam(nullPref,
                   addParam(localPolicy,makeHPL)))))));
```

```
          (t !impName !iref ?res: Message;
           Importer[ g,t...](iid,iref,tref,known,...)
           []
           t !impName !iref ?irefs: IRefSet;
           Importer[ g,t...](iid,iref,tref,(known Union irefs),...))
      ...
      []
      g ... (* operations invoked on "known" interface set *)
    endproc (* Importer *)
```

We consider the modelling of the import request parameters in turn. As with the export request, we assume that there exists some means whereby importers can be identified. The import request actually creates the set of operations that the importer is interested in here, *i.e.* operations X_1 and X_3. It might well be the case that the set of operations required is given by a reference to some particular set of operations. We note here that this approach to specifying the sets of operations that are required opens up numerous issues that are likely to be addressed by type management systems. For example, it might be the case that no one interface supports all of the operations that the importer requires, but the operations can be found in more than one, possibly remote, interfaces. This phenomenon of type checking based on particular operations in an interface is termed F-bounded polymorphism [39]. Whether it is meaningful to return more than one interface reference to an importer for subsets of the requested operations depends to a great extent on the importers and the semantics attached to the operations requested. For example, the operations may be dependent upon one another. If this is the case then it may not be possible to separate them.

We specify that the importer is interested in service properties related to resource availablity, *i.e.* the values of the available resources are to be returned with the matching offers. We note that this, as in the case of the exporter specified previously, might be null. The importer also wishes to know the expiry dates associated with exported offers, *i.e.* how long the service is likely to be available.

The matching constraints are represented by the values *maxCost* for cost constraints and *minDate* for expiry date. Thus the trader will only return interface references for services that have an associated cost less than *maxCost*, have an expiry date longer than *minDate*, and support the operations X_1 and X_3.

For simplicity we do not deal with issues relating to ordering the matching service offers, *i.e.* we have a null selection (*nullPref*). We omit the LOTOS text for brevity.

The import policy can take many different forms with many different levels of complexity. For example, import policies might be based only on searching the local trader. We model this through the value *localPolicy*. Alternatively, all traders that this trader knows about might be searched also. These are perhaps the two most simplistic policies. Other ones are possible. For example, if the trader was charging users for services then import policies might be based on cost limits for using the trader. It might be the case that importers request the search to stop after a certain amount of time if timeliness constraints are more important.

8.1.7 A More Complete Trader Specification

>From the specification of the importer and exporters we may expand upon the initial outline for the trader specification given previously. A more complete structure for a trader specification might be:

```
   process trader[ t ](tref: IRef, exports: expOffers):noexit:=
      t ?op: Name !tref ?pl: HOPList;
      ( [ not(makeOp(op,pl) IsIn getOps(tref)) ] ->   ... (* return error *)
        []
        [ (makeOp(op,pl) IsIn getOps(tref)) and (op eq expName) ] ->
           t !op !getIRef(pl) ?id: ID[ not(id IsIn getIds(exports))]
           trader[ t ](tref,addOffer(id,getIRef(pl),getSP(pl) Union getSOP(pl),exports))
        []
        [ (makeOp(op,pl) IsIn getOps(tref)) and (op eq impName) ] ->
           t !op !getIRef(pl)
                 !search(getSP(pl) Union getSOP(pl),getOps(getIRef(pl)),exports);
           trader[ t ](tref,exports)
        |||
        trader[ t ](tref,exports))
endproc (* trader *)
```

For simplicity we avoid dealing with all of the checks that might be made for erroneous invocations, *e.g.* where the export and import operations do not have correct parameters.

Here the sort *expOffers* represents the sets of service offers that have been successfully exported to the trader. This may be represented as:

```
type ExportedOffers is IdSet, IRefSet, Constraints
     sorts expOffers
     opns empty   : -> expOffers
          addOffer: Id, IRef, Constraints, expOffers -> expOffers
          remOffer: Id, expOffers -> expOffers
          getIds  : expOffers -> IdSet
          search  : Constraints, OpSet, expOffers -> IRefSet
endtype (* ExportedOffers *)
```

We omit the equations here for brevity. Operation *getIds* returns the identifiers of all exported offers. We are ensured that all exported offers are uniquely identified through the use of a selection predicate in the process algebra. Operation *remOffer* removes an offer from the exported offers that has a particular identifier. Operation *addOffer* requires as input: an identifier, an interface reference and a set of constraints. These constraints are then assigned to the constraints that are associated with the interface reference.

We also provide an operation for searching through the service offers that have been exported to the trader. This operation allows searches to be made based on operations associated with exported offers and the constraints that they have attached to them. The equations associated with this operation could well be very complex depending on the constraints and operations that have been supplied. For example, the search might not find offers with all of the operations supplied even though they satisfy the constraints. Alternatively, the search might find offers that possess all of the operations requested but that do not satisfy all of the constraints that have been supplied. Other possibilities exist also.

8.2 Conclusions

In this section we have seen how LOTOS can be applied to develop a trading system where services can be found based on static or dynamic features that are associated with them. In addition we have seen how references to these services can subsequently be used to access the services.

Related work [112, 149] has focussed on adding new mobility features to the LOTOS language so as to enable dynamic communication paths to be established. We have shown that this is not absolutely necessary. The LOTOS language is expressive enough for certain degrees of dynamism to be supported. We note here though that our approach does have certain limitations. The primary problem with this approach is that all processes that have the possibility to interact with one another at a common gate must participate (synchronise) on all events at this gate. This includes synchronising on events where the process was not expressly referenced, *i.e.* the interface reference associated with the event offer referenced a different process.

It is quite possible to extend the approach given here. For example, it is possible to extend the constraints that might apply to service properties. One such extension might be the inclusion of location constraints, *e.g.* import requests might specify that the services of interest should exist within a specific location. In the previous section we largely avoided dealing with locations and how they can be used to influence the importing of offers. It might well be the case that location plays a critical part in offer selection though. For example, importers might specify that only services at a particular location are to be returned. Alternatively, it might be the case that service locations can have some means whereby their nearness (or remoteness) can be used as a basis for deciding on which service is selected. Such extensions could be specified, *e.g.* through data types modelling locations with operations that can check on the equality of locations or distance between locations. A simple example of how this might be represented is:

```
type Location is NaturalNumber, Boolean
     sorts Loc
     opns locX, locY, locZ: -> Loc
          distanceBetween : Loc, Loc -> Nat
          Ord             : Loc -> Nat
     eqns ...
endtype (* Location *)
```

We omit the equations for brevity. Locations might simply be represented by sorts and operations exist that allow comparisons between these sorts. Thus the more remote an interface is from another the greater the difference between the operation *Ord* when applied to these interfaces. An importer might use this information when importing from a trader. For example, constraints of the form 'return only those interface references for services within a certain distance', *i.e.* the difference in value between the importer location and the location of the service registered with the trader.

The notion of type checking is fundamental to trading. Most current approaches to type checking in distributed systems are based solely on signatures [172]. Further, the checks that they perform with regard to establishing type equivalence is based on the classical type theory of subsetting operations, contravariance of inputs and covariance of outputs [40]. We have seen that even at the syntactic level, *i.e.* without considering issues related to semantics, this need not necessarily be the case. Type checking for trading should provide as much flexibility as possible for the trader importers, or at least as much flexibility as the importers demand. The approach given here and in chapter 7 has shown how useful type relationships can be established in different ways. Thus import requests for sets of operations not available at any one interface should not automatically be simply rejected, *i.e.* error messages stating that no matching offers were found. Further, issues related to type checking the constraints that might be attached to interfaces should also be dealt with, *e.g.* checking that constraint values supplied are compatible with the constraints associated with the interface.

8.3 Specification of a Producer and Consumer Flow Configuration

As discussed, ODP introduces concepts necessary for modelling and reasoning about multimedia flows of information. We have seen already how Z could be used to reason about issues such as maximum throughputs and delays associated with these flows. In this chapter we consider in more detail how Z can be applied to specify a configuration of producer and consumer information flows. In particular we show how Z can be used to model the actual flow of information and changes to the flow of information. In addition we show how the non-functional aspects of the producers and consumers are used to restrict the flows and flow manipulations that can occur.

To enable a higher level of prescription to be given we deal with specific operations on the flow of information. We consider operations for starting and stopping the flow and operations to increase and decrease the rate of flow production.

For simplicity we assume that the data contained in the information flow from the producer is that required by the consumer. That is, we avoid dealing with issues related to type checking of multimedia flows of information. Rather, we focus only on how the information flows are passed from producer to consumer in such a way that the temporal constraints imposed on the flows are satisfied. Another simplification we make is to deal only with producers and consumers of single flows.

8.3.1 Background Concepts for Modelling Information Flows

As presented in section 5.1.2, an information flow can be represented in Z by a schema consisting of a sequence of frames. In section 5.1.2, we considered a frame as a schema consisting of a time stamp, some data and a label. Previously we considered the time stamp as a natural number. In this section we consider time as a real number. This allows us to have a dense model of time and also highlights the versatility of the Z language. We model a frame here as:

$$
\begin{array}{|l}
\textit{Frame} \\
\hline
ts : \mathbb{R} \\
data : Data \\
label : Name \\
\hline
\end{array}
$$

Information flows associated with producers and consumers of information may be either flowing or stopped. We introduce the following free type to represent the possible status that may be associated to the information flows attached with producers and consumers.

$$Status ::= Stopped \mid FlowIn \mid FlowOut$$

8.3.2 Formalising the State of Producers and Consumers

Interfaces that produce or consume interface flows are required to access variables modelling time. In section 5.1.2 we showed that time could be represented in Z in several ways. We consider here a local model of time where the producers and consumers of information flows have local variables modelling the current local time. The local time is incremented when operations occur, *i.e.* information items are sent or received. The amount that the time is incremented by depends on the rate of production or consumption of flows. Since we are interested in modelling bursty flows here, *i.e.* flows that may have their rates of production or consumption varied, we model the rate of production or consumption of flows by a variable local to the producer or consumer.

We may model a producer state by the following Z fragment:

PState

$$
\begin{array}{l}
ps : \operatorname{seq} Frame \\
ptnow : \mathbb{R} \\
pstat : Status \\
prate : \mathbb{R} \\
pCons : ProducerCons \\
\hline
\forall f_1; f_2 : Frame \mid (f_1; f_2) \in ps \bullet f_2.ts > f_1.ts \land \\
pstat \in \{Stopped; FlowOut\} \land \ldots
\end{array}
$$

Here we state that the producer state is given by a sequence of frames (*ps*), the current time (*ptnow*), the rate of production of frames (*prate*, *i.e.* the time it takes for a producer to produce a frame), the current status (either stopped or flowing out from the interface), and a set of constraints associated with the producer information flow. We also state that all frames in the sequence of frames to be sent should have increasing time stamps. Here the dots are dependent upon the constraints associated with the multimedia flow and how they influence the flow itself. To allow these predicates to be given, it is necessary to be more prescriptive regarding the constraints of interest. We consider the maximum delay, maximum throughput and (bounded) jitter associated with the information flow. These constraints may be represented as:

ProducerCons

$$
\begin{array}{l}
pmt : \mathbb{R} \\
pmd : \mathbb{R} \\
pjb : \mathbb{R} \times \mathbb{R}
\end{array}
$$

We note here that it is normally the case that Z specifications are written in a definition before use manner. This is especially the case if tool support is used. Semantically the ordering is unimportant provided the Z text can be ordered in such a way that all Z fragments are defined before they are used.

In the schema *ProducerCons*, *pmt* represents the producer's maximum throughput, *pmd* the maximum delay and *pjb* the upper and lower limits on jitter. We note here that we do not concern ourselves with how these values are calculated and set. We are only interested that these values can be set. We shall see shortly that the values can be assigned non-deterministically. With this, a more complete model of the producer's state is given by adding the predicates:

$$
\begin{array}{l}
\forall f_1; f_2 : Frame \mid (f_1; f_2) \in ps \bullet \\
\quad f_2.ts - f_1.ts < pCons.pmd \land 1 \operatorname{div} prate < pCons.pmt
\end{array}
$$

Here we state that all consecutive frames in the sequence should have time differences less than the maximum allowed delay. We also state that the rate (*prate*) of production of frames must be less than some upper throughput limit. For simplicity we assume that throughput is measured over a single time unit. Extending this to cover other ranges of time can be achieved using the approach given in section 6.3.4. We note here that whilst these constraints can be asserted directly and used to influence the production of frames, it is not possible to write explicit predicates on the jitter. The jitter depends on the actual sending and receiving of frames. Before we show how jitter constraints can be used to influence the information flow, we consider the consumer state.

We may represent consumer states similarly. This can be represented as:

$$
\begin{array}{|l}
\textit{CState} \\
\hline
cs : \mathrm{seq}\ \textit{Frame} \\
ctnow : \mathbb{R} \\
cstat : \textit{Status} \\
crate : \mathbb{R} \\
cCons : \textit{ConsumerCons} \\
\hline
\forall f_1; f_2 : \textit{Frame} \mid \langle f_1; f_2 \rangle \in cs \bullet f_2.ts > f_1.ts \land \\
cstat \in \{\textit{Stopped}; \textit{FlowIn}\} \land \ldots \\
\hline
\end{array}
$$

For brevity we do not explain the variables *ctnow*, *cstat* and *crate* which are the consumer equivalent of the variables given for the producer. The variable *cs* represents the frames that have been received by the consumer. For simplicity we assume that frames arrive in the order in which they are sent. This allows us to assert that all frames in the sequence have increasing time stamps. This need not necessarily be the case though, *i.e.* frames may arrive in an order different from that which they were sent. It is quite possible to extend the approach given here to deal with such cases.

As above, the dots are dependent upon the specific constraints a consumer might have on an information flow. For simplicity, we assume the consumer has the same constraints as the producer, *i.e.* on maximum delays, throughputs and jitter bounds. This can be represented as:

$$
\begin{array}{|l}
\textit{ConsumerCons} \\
\hline
cmt : \mathbb{R} \\
cmd : \mathbb{R} \\
cjb : \mathbb{R} \times \mathbb{R} \\
\hline
\end{array}
$$

Here *cmt* represents the consumer's maximum throughput, *cmd* the consumer's maximum delay and *cjb* the upper and lower time limits at which a frame is acceptable. With this, a more complete model of the consumer's state is given by adding the predicates:

$$
\begin{array}{l}
\forall f_1; f_2 : \textit{Frame} \mid \langle f_1; f_2 \rangle \in cs \bullet \\
\quad f_2.ts - f_1.ts < cCons.cmd \land 1 \operatorname{div} crate < cCons.cmt
\end{array}
$$

Here we state that all consecutive frames in the consumer flow should have a time difference less than the maximum allowed delay. We also state that consumption of frames is less than some maximum throughput value. As with the production of flows, jitter constraints are considered when we deal with the sending and receiving of information flows.

As done in section 7.5.2 we may represents the constraints that might be associated with a producer or consumer generally by a parameterised free type definition. This can be represented as:

$$
\textit{Constraints} ::= \textit{prodBindCons}\langle\textit{ProducerCons}\rangle \mid \textit{consBindCons}\langle\textit{ConsumerCons}\rangle
$$

Similarly we represent the satisfaction of constraints associated with producers and consumers as:

$$
\textit{Satisfies} : \textit{Constraints} \leftrightarrow \textit{Constraints}
$$

For brevity we omit the associated predicates. The satisfaction of throughput and delay constraints were given in section 7.5.2. Satisfaction of jitter constraints requires that the producer allowed jitter is within the consumer jitter range.

To model the instantiation of the producer and consumer we use the Z convention [161] of considering only the primed state schemas, *i.e.* we do not care what the state was before the producer and consumer were initialised. The initialisation of the producer and consumer may be represented as:

$$
\begin{array}{|l}
\textit{PCInit} \\
\hline
\textit{PState}' \\
\textit{CState}' \\
\hline
\exists pc : \textit{ProducerCons};\ cc : \textit{ConsumerCons};\ ss : \mathrel{seq} \textit{Frame} \mid ss \neq \langle\rangle \bullet \\
\quad (\textit{consBindCons}(cc); \textit{prodBindCons}(pc)) \in \textit{Satisfies} \land \\
\quad (\textit{prodBindCons}(pc); \textit{prodBindCons}(pCons)) \in \textit{Satisfies} \land \\
\quad (\textit{consBindCons}(cc); \textit{consBindCons}(cCons)) \in \textit{Satisfies} \land \\
\quad pCons' = pc \land cCons' = cc \land ps' = ss \land \\
\quad cs' = \langle\rangle \land ptnow' = ctnow' \land \\
\quad pstat' = cstat' = \textit{Stopped} \land \\
\quad prate' = 1 \mathrel{div} pc.pmt \land crate' = 1 \mathrel{div} cc.cmt \\
\hline
\end{array}
$$

This states that there exist producer and consumer constraints such that:

- the values that the producer and consumer constraints are set at is satisfactory to the consumer and producer respectively, *e.g.* the producer's maximum delay is less than the maximum delay that the consumer can tolerate.

- the values at which the producer's constraints are being set are satisfactory to the producer, *e.g.* the throughput is not being set at a value that the producer cannot produce.

- the values at which the consumer constraints are being set are satisfactory to the consumer, *e.g.* the jitter is being set at values within the upper and lower jitter limits.

These constraints (if they exist) are then assigned to the producer and consumer constraints. We also state that the producer is instantiated with a non-empty sequence of frames to send and the consumer initially has received no frames. The producer and consumer local times are synchronised and both have the current status of stopped. Finally we state that the producer and consumer rates are set to values given by the reciprocal of the maximum throughputs.

We note that we define a single relation between constraints here. In reality this is likely to be split into several different relations since satisfaction of constraints is different for producers and consumers, producers and producers, and consumers and consumers. These differences were discussed in sections 7.5.2 and 7.5.5.

8.3.3 Formalising Interfaces for Controlling Information Flows

As well as having interfaces for producing and consuming information flows, producers and consumers of information flows are likely to have operations for manipulating these information flows. We term these *control* interfaces. It is likely that these interfaces will follow an RPC-like mode of interaction, where messages are sent and responses are given to these messages. For simplicity here, we assume a basic set of messages that are passed between producers and consumers. These may be represented by the free type definition:

$$
\begin{aligned}
\textit{Message} ::= {} & \textit{ExceedsMax} \mid \textit{AlreadyStopped} \mid \textit{AlreadyStarted} \mid \\
& \textit{NotStarted} \mid \textit{ArrivedTooEarly} \mid \textit{ArrivedTooLate} \mid \textit{Ok} \mid \ldots
\end{aligned}
$$

The dots here signify that these messages may well be extended to cover numerous other situations. We shall discuss some of these other situations at the end of this chapter.

Starting and Stopping a Producer Flow

The most elementary of operations are the starting and stopping of flows. A flow being started successfully may be represented by:

ProdStartFlowOk

$\Delta PState$
$res! : Message$

$pstat = Stopped \land ps' = ps \land$
$ptnow' = ptnow \land pstat' = FlowOut \land$
$prate' = prate \land pCons' = pCons \land res! = Ok$

The preconditions for this operation require that the flow is currently stopped. The postconditions state that the status of the producer is changed to *FlowOut*. All other variables associated with the producer state are unchanged. It might well be the case that the occurrence of this operation schema might take a non-negligible time. In this case, the local variable modelling time should be incremented by some amount. We assume that the operation takes zero time and that time is only incremented on the sending of frames.

A start flow operation may fail if the producer is already producing the flow. An unsuccessful request to start a flow may be represented by:

ProdStartFlowNOk

$\Xi PState$
$res! : Message$

$pstat = FlowOut \land res! = AlreadyStarted$

Here we say that the state of the producer is unchanged. The total operation to start a flow may be represented by:

$$ProdStartFlow \triangleq ProdStartFlowOk \lor ProdStartFlowNOk$$

For brevity we do not provide the operations for stopping a flow. These are similar to the starting of a flow with the obvious changes to the predicates. For reasons of brevity we also do not provide the operations for starting or stopping a consumer flow. These are similar to the operations given for starting and stopping producer flows.

Increasing and Decreasing the Rate of Production

The rate of production of information flows may be increased or decreased depending on the demands of the consumer. A successful operation to increase the rate of flow from the producer may be represented by:

ProduceFasterOk

$\Delta PState$
$nr? : \mathbb{R}$
$res! : Message$

$pstat = FlowOut = pstat' \land nr? < prate \land pCons{:}pmt \geq 1 \operatorname{div} nr? \land$
$ps' = ps \land ptnow' = ptnow \land prate' = nr? \land pCons' = pCons \land res! = Ok$

This operation requires that the producer is currently outputting information and that the new rate, *i.e.* the time to produce frames, is less than the existing one but within the throughput limits. The postconditions state that the status, current time, sequence of frames and constraints are unchanged. An output message is sent stating that the operation was successful.

We consider two cases in which this operation might fail. The operation might fail if the requested rate would result in the maximum throughput constraints being exceeded. This can be represented as:

ProduceFasterExceedMax

$\Xi PState$
$nr? : \mathbb{R}$
$res! : Message$

$pCons{:}pmt \geq 1 \operatorname{div} nr? \land res! = ExceedsMax$

Here the state of the producer is unchanged if the rate exceeds the maximum allowed throughput. An appropriate error message is also returned.

A request to produce a flow faster may also fail if the producer is not currently producing a flow, *i.e.* it has not yet been started. This can be represented as.

ProduceFasterNotStarted

$$\Xi PState$$
$$nr? : \mathbb{R}$$
$$res! : Message$$
$$pstat = Stopped \land res! = NotStarted$$

The total operation showing how the producer flow can have its rate increased is given as:

$$ProduceFaster \widehat{=} ProduceFasterOk \lor ProduceFasterExceedMax \lor ProduceFasterNotStarted$$

We omit the operations showing how the information flow rates can be reduced. These are similar to those given for the increase in production but with the obvious predicate changes, *e.g.* checking that the rate does not become zero or negative. For reasons of brevity we also do not provide the operations for increasing or decreasing the consumer consumption rate. These are similar to the operations given for changing the producer flow rate but with the predicates modified accordingly.

8.3.4 Formalising the Production and Consumption of Information Flows

Given that we have taken a local model of time, the production of frames requires that the frames are time stamped with the local time when they are to be sent. The sending of a frame from a producer may be represented as:

PSendFrame

$$\Delta PState$$
$$f! : Frame$$
$$ps \neq \langle\rangle \land pstat = FlowOut = pstat' \land$$
$$ps' = tail\ ps \land prate' = prate \land pCons' = pCons$$
$$(\exists timesent : \mathbb{R} \mid (ptnow + prate - first\ (pCons{:}pjb)) \leq timesent \land$$
$$timesent \leq (ptnow + prate + second\ (pCons{:}pjb)) \bullet prate' = timesent \land$$
$$(\textbf{let}\ nextframe == (\ulcorner Frame \mid data = (head\ ps){:}data \land$$
$$ts = timesent \land$$
$$label = (head\ ps){:}label) \bullet f! = nextframe\ ))$$

Several things should be pointed out here. Firstly, we require that the producer has frames to send. The producer should currently have the status set so that it is producing frames. Sending a frame removes that frame from the sequence of frames to be sent. The current rate and constraints associated with the flow are unchanged. The actual time at which the frame was sent is dependent upon the current rate, the current time and the jitter. The time the frame is sent is given by the current time plus the current rate and within the jitter bounds.

The actual frame sent is the head frame in the sequence of frames to be sent. This is time stamped with the value for the time calculated previously. We note here that the use of the definite description requires that a proof obligation is fulfilled to ensure that the frame sent is unique. This obligation is satisfied through modelling all frames in the sequence with increasing (*i.e.* non-equal) time stamps. Since no frame in the sequence has the same time stamp, the frame sent with the current time is unique. Also we require as a precondition that the sequence of frames is non-empty.

A consumer may receive a frame successfully provided the constraints for its acceptance are satisfied.

CGetFrameOk

$$\Delta CState$$
$$f? : Frame$$
$$cstat' = FlowIn = cstat \land crate' = crate \land cCons' = cCons \land$$
$$ctnow - first\ (cCons{:}cjb) \leq f?{:}ts \leq ctnow + second\ (cCons{:}cjb) \land$$
$$cs' = \langle f? \rangle \frown cs \land ctnow' = ctnow + crate$$

The preconditions for this operation are that the consumer can currently receive frames, *i.e.* its status is currently *FlowIn*. The operation itself does not affect the rate of consumption, the current status or consumption constraints. The frame is only accepted provided the time stamp is within the allowed jitter range associated with the consumer constraints. If the time constraints are satisfied then the frame is accepted and added to the sequence of frames received and the current time is incremented by whatever value the rate is currently set at, *i.e.* how long it takes to add the frame to the sequence of received frames[3].

We consider two situations in which a frame may be rejected by a consumer. Firstly, the frame may arrive outside the time constraints, *e.g.* too early or too late. This can be represented as:

$$
\begin{array}{|l}
\textit{CGetFrameTimeNOk} \\
\hline
\Xi \textit{CState} \\
f? : \textit{Frame} \\
\textit{res}! : \textit{Message} \\
\hline
(f?.ts \geq (\textit{ctnow} + \textit{second}\ (\textit{cCons.cjb})) \land \textit{res}! = \textit{ArrivedTooLate}) \lor \\
(f?.ts \leq (\textit{ctnow} - \textit{first}\ (\textit{cCons.cjb})) \land \textit{res}! = \textit{ArrivedTooEarly}) \\
\hline
\end{array}
$$

For simplicity here we assume that the state of the consumer is unchanged, *i.e.* that the operation takes no time to occur.

A frame might also be rejected if the consumer is not yet ready to start consuming frames, *i.e.* it is currently stopped. This can be represented as:

$$
\begin{array}{|l}
\textit{CGetFrameNotStarted} \\
\hline
\Xi \textit{CState} \\
f? : \textit{Frame} \\
\textit{res}! : \textit{Message} \\
\hline
\textit{cstat} = \textit{Stopped} \land \textit{res}! = \textit{NotStarted} \\
\hline
\end{array}
$$

The actual sending of a frame from a producer to a consumer may be represented as:

$$
\begin{aligned}
\textit{ProdSendtoConsumer} \triangleq\ & \textit{PSendFrame} \land (\textit{CGetFrameOk}[f! = f?] \lor \\
& \textit{CGetFrameTimeNOk}[f! = f?] \lor \\
& \textit{CGetFrameNotStarted}[f! = f?])
\end{aligned}
$$

8.3.5 Specifying Latency and Imperfect Communications Mediums

It is possible to specify latency simply through a schema that accepts a frame and outputs that frame with some increased time stamp. This can be represented as:

$$
\begin{array}{|l}
\textit{Latency} \\
\hline
f?; df! : \textit{Frame} \\
\hline
\exists \textit{delay} : \mathbb{R}_{\bullet} \\
df!.ts = f?.ts + \textit{delay} \land df!.data = f?.data \land df!.label = f?.label \\
\hline
\end{array}
$$

Here the delayed frame (*df!*)is increased by some value non-deterministically. The latency between the sending of a frame from a producer to its arrival at the consumer may be represented as:

$$
\begin{aligned}
\textit{ProdSendtoConsumerwithLatency} \triangleq\ & \textit{PSendFrame} \land \textit{Latency}[f! = f?] \land (\textit{CGetFrameOk}[df! = f?] \lor \\
& \textit{CGetFrameTimeNOk}[df! = f?] \lor \\
& \textit{CGetFrameNotStarted}[df! = f?])
\end{aligned}
$$

It is possible to specify imperfect communications mediums in Z. This can be achieved for example by schemas that accept frames as inputs and do not give frames as outputs. We note here though that this notion of erroneous behaviour is limited in the sense that there is no real notion of spontaneity as might be found in LOTOS, say, with the internal event **i**. This model of imperfect communications simply represents a choice in the actual behaviour that the system can exhibit.

[3] Since we regard the time for checking the time associated with an arrived frame as negligible.

8.4 Conclusions

In this section we have seen that Z can be used quite successfully for modelling and reasoning about information flows. We have seen that the language allows manipulation of flows and checking constraints that might apply to flows directly.

The work here may well be extended. For example, it is possible to model dynamic changes to the constraints that apply to the information flows. This might involve the renegotiation of the constraints that are set initially when the flows were instantiated, *e.g.* when a consumer can no longer consume as quickly for whatever reason.

The approach given here has to a certain extent avoided the problems involved in using Z to specify object based systems. That is, we have not focussed on issues related to specifying interfaces and the checks prescribed by ODP for the legal composition of interfaces. Rather, we have used Z in a more classical style. Thus producers and consumers of information flows are simply conjoined. On the one hand this allows a straightforward specification approach to modelling information flows. Unfortunately this approach does not allow any form of *a priori* checking in the specification itself as to whether the composition is likely to be successful on a syntactic level. We are, however, able to specify that the constraints associated with the information flows must be satisfied.

We note also that as discussed in section 5.1.2, the Z notation is essentially a flat notation. There is as such no real ordering of operations, *e.g.* in the sense that the producer sends a frame following which a consumer consumes the frame. The whole action of production and consumption (even with latency) is a single operation in Z, *i.e.* it happens in its entirety or not at all. Despite this limitation we have seen that Z can be used effectively to model flows of information.

Chapter 9

Conclusions

This chapter reviews the goals of this thesis and discusses how successful our investigation of these goals has been. The chapter also identifies some possible future developments for the work.

9.1 Summary of the Thesis

The goals of this thesis as put forward in chapter 1 were to investigate how the writing of specifications of distributed systems could be alleviated through adopting an architectural approach. In this thesis we based our specification fragments on the ODP-RM architecture. In particular we focussed on two areas of ODP: the basic modelling concepts, specification concepts and behavioural concepts contained in Part 2 [108], and the more prescriptive viewpoint languages of Part 3 [108] with particular emphasis on the computational viewpoint.

The computational viewpoint deals with objects interacting at interfaces. Objects can only interact with one another provided they have some common understanding of the services (or behaviours more generally) offered at these interfaces. This understanding has typically been based on the syntactic aspects of interfaces. This is limiting since syntax alone is unlikely to contain sufficient information to enable interfaces to decide whether they are able to access other interfaces. We have thus investigated how additional information in the form of constraints can be used as a basis for increasing the knowledge about behaviour offered at interfaces. Specifically, we have focussed on timing constraints, cost constraints and resource constraints. We note here that there exists a duality in the openness and heterogeneity of systems regarding the syntactic aspects of types. For example, interface definition languages allow for the separation of syntactic aspects from implementation (behavioural) considerations. On the one hand this promotes heterogeneity, *i.e.* users are not tied down to particular operating systems, languages etc[1]. Treating systems purely from syntactic considerations is, however, detrimental to openness. Syntactic treatment of systems does not provide enough information to enable the issues related to openness to be addressed. As argued in section 1.1.1, syntactic aspects of systems are primarily an initial guideline as to whether those systems can interwork.

Ideally of course, type systems should deal with behavioural and non-functional aspects of systems if they are to interwork correctly. We have seen how LOTOS and Z can be used to specify the non-functional constraints that might be associated with a system in a form that is directly checkable. Unfortunately, it is not the case that LOTOS or Z allows for behavioural type checking in the form that might be required in a distributed system, *i.e.* where a resource is discovered and expected to interwork with an existing system "on the fly". We note that it is possible to specify such issues in Z, but implementing them is likely to be undecidable.

As well as traditional message passing between objects, ODP attempts to deal with objects supporting information flows. Flows of information place stringent timing constraints on the interfaces with which they are associated if the interfaces are to pass the information successfully. We have investigated how

[1] Provided of course that language bindings have been devised for the user's preferred language and the interface definition language can be understood on that operating system.

LOTOS and Z can be used to model such constraints. We presented a specification of a producer and consumer flow configuration that highlighted in detail how Z could be used to specify and check timing issues.

Distributed systems are inherently dynamic. They offer the user a means to find new services and information that might exist in the system, and interact with them dynamically. To enable such dynamic issues to be addressed we have focussed on applying LOTOS in such a way that interface references can be treated as first class citizens. These models of interface references were applied to two main areas: dynamic binding and substitution. Both of these areas required that interface references be decomposed. This decomposition should permit checks on operations at the interfaces and any constraints that might be associated with the interfaces. We have shown how LOTOS can be used to address both of these issues. A trader specification was presented that showed how LOTOS could be used to model this dynamism.

9.2 Comparison of LOTOS and Z for Specifying Distributed Systems

In chapter 4 we argued that LOTOS and Z were languages from different ends of the formal language spectrum. As a result, one would have expected the development of specification architectures using these languages to be radically different. Certainly we have seen that LOTOS and Z have different advantages and disadvantages for specifying distributed systems. For example, Z allows global predicates to be given that can be used to influence and restrict behaviours. LOTOS on the other hand is well suited for describing collections of actions forming complex behaviours.

In our work here we have attempted to show how these features can be incorporated into LOTOS and Z. For example, in our abstract Z approach we have modelled behaviour through graphs of actions that is similar to the representation of behaviour in the process algebra part of LOTOS. In LOTOS, we have introduced a form of type checking on behaviours that includes constraint checking. To a certain extent, these constraints represent statements about behaviour that are similar to the global predicates that might be found in Z. Here the predicates are associated with processes though. We note that for these constraints to be meaningful, the process behaviour that the constraint is attached to should realise (satisfy) the constraint. For example, if a process has a constraint that it can accept a certain number of invocations then the behaviour exhibited by this process should reflect this.

Thus in this thesis, we have used LOTOS and Z in a way that allows them to capture the advantages in the other language, *i.e.* Z or LOTOS respectively, whilst at the same time keeping their own inherent advantages. As a result the striking comparison of approaches has, to a great extent, not materialised.

9.3 Contributions of the Thesis

This thesis has made numerous contributions to several areas. These areas include:

- distributed systems in the large;
- formal models of object-based distributed systems;
- formal models of multimedia and time-dependent systems generally.

Regarding the first bullet point, it is widely acknowledged in distributed systems that type checking based entirely on syntactic elements is limited. We have shown how it is possible to extend type checking with assertions (constraints) that can be checked directly when binding or substituting systems. Whilst not guaranteeing that two systems will interwork correctly, this approach can be seen as a useful step forward in checking that systems can interwork correctly.

We have identified the different requirements that are placed on type checking mechanisms when causalities are introduced into distributed systems. In addition, we have shown that type checking when substituting interfaces requires different checks from those required for binding interfaces. This includes the syntactic checks as well as the checks on constraints.

This thesis has considered how object-based concepts can be applied to develop specification architectures for distributed systems. Object-based specification approaches have been a well investigated area

[44, 147, 81, 136]. Few approaches have considered how object-based ideas and formal techniques can be applied to distributed systems however. As discussed in section 2.2.4, object-orientation is a useful modelling paradigm, but not all of the object-oriented conceptual baggage is necessarily required when describing (specifying) distributed systems. Rather, we argue that specification of distributed systems should focus more on issues related to:

- evolution: specifying how one system may be replaced by another and the kind of checks that are required. In particular, evolution in active systems is of interest, *i.e.* where the system is already interworking with other systems.
- discovery: specifying systems so that they can receive information that they can use to decide whether they wish to enter into new communications.

We have seen how LOTOS can be used in such a way that it allows both of these areas to be dealt with. We note that LOTOS is limited to a certain extent when specifying true dynamism, where new[2] communication channels are established between interacting objects. LOTOS models interaction points through gates. The gates assigned to a process are given statically. It is not the case that these gates may be passed around to set up new interaction points between processes. This facility is only available in certain process calculi, *e.g.* the Pi-calculus [143]. Nevertheless we have seen in chapter 8 how a form of dynamism can be achieved through using the synchronisation features of LOTOS. That is, through processes synchronising at a common gate and using the action denotations to limit the processing of the messages sent. This has the disadvantage that all processes wishing to receive or send these "broadcast" messages through a certain gate must receive all messages sent through that gate.

Our work has also contributed to formal models of multimedia and time-dependent systems more generally. Although LOTOS provides a model of the temporal ordering of actions, we have seen how the language can be used in such a way that it allows time-dependent systems to be specified directly. Our attention was focussed on aspects of multimedia but the same approach could be applied more generally. There have been several approaches to modelling time in LOTOS [44]. In standard LOTOS the approach has been to represent time by events associated with some form of clock process. As discussed in section 5.1.2 this allows for an abstract model of time to be introduced into the specification, but it is often the case that real-time systems, *e.g.* multimedia systems, require a more definite model of time. As a result, many different works have extended LOTOS with timing facilities [112, 139, 136]. We have shown that the language is more flexible than frequently claimed. Also, the modelling of time in LOTOS as opposed to a time extended version of LOTOS allows us to work in a standardised formal technique. This has several advantages including the lack of learning curve in learning a new time extended language as well as being able to use existing tool support developed for LOTOS.

In contrast to the large amount of work on time extended process algebras, there have been few approaches to modelling time in Z [70, 117]. It could be argued that the main reason for this is that timing considerations are trivial to achieve in Z. It is possible that the lack of documented approaches to modelling time in Z stem from this triviality. Our work documents several different approaches to modelling time in Z, and as such can be regarded as contributing to the field of specification techniques for modelling real-time systems.

9.4 Discussion

Given that we have argued that an architectural approach can help alleviate many of the problems in writing specifications of distributed systems, we discuss briefly whether this has really been the case.

Part 2 of the ODP-RM [108] provides generic concepts that can be useful when modelling, specifying, and generally reasoning about distributed systems. Due to their genericity, these concepts may only ever be formalised through a semi-informal approach. For example, it is not possible to formalise what the concept of *behaviour* is in any formal language to a degree where the formalisation can be used and applied directly. As a result, the "formalisation" of certain concepts can only be achieved through informal modelling suggestions.

[2] In the sense of a connection between objects (processes) that never existed previously.

In the computational viewpoint of Part 3 of the ODP-RM [109] we have seen that there is a higher level of prescriptivity in the definition of concepts. As a result, the development of specification fragments suitable for architecting specifications of distributed systems has shown more potential. Unfortunately, whilst it is the case that more prescription is given, the prescription does not lend itself directly to the specification languages LOTOS and Z. The problem with Z is that the language suffers from a lack of object modelling facilities. Thus although it is possible to specify collections of specification fragments representing the more prescriptive concepts given in the computational viewpoint, Z in its classic usage does not lend itself to specifying behavioural aspects associated with these structures. As a result, an approach was presented that allowed behaviour to be associated with these specification fragments in a more abstract manner. On the one hand this provided a means to specify (powerful) relations between the behavioural aspects of interfaces. On the other hand, it could be argued that the approach given was not using Z as the language was designed. Further, although it was possible to specify complex relations between interfaces (or sets of interfaces) that satisfied and extended the rules given by ODP, the specification fragments were not themselves immediately re-usable.

In comparison, LOTOS has more potential for modelling object based systems. With LOTOS, however, the prescription given in the computational concepts and structuring rules of ODP was not directly representable in facilities of the language itself. That is, the prescription given was largely syntactic in nature, *e.g.* interfaces being bound should have similar operation names. Given that we model interfaces and behaviour in the process algebra, and that binding corresponds to establishing synchronisation possibilities between processes, it is not a natural feature of the process algebra that syntactic checks are enforced when synchronisation is done. That is, gates are used for synchronisation and any process may be synchronised with any other process. Whether they deadlock, livelock or have any other arbitrary behaviour is typically not considered in LOTOS. Such checks are normally performed outside the specification, *e.g.* when testing is done. As a result our approach has imposed a syntactic layer on the synchronisation features of LOTOS. Although this provides us with a workable solution, there is a danger of working solutions and good specification practice becoming separated. The Act One approach developed in chapter 7 highlights this. To overcome the limitations of LOTOS in its classic usage for specifying and checking type relationships, it was necessary to use the language to describe itself. As a result, we can see that the language can be used in a stylised (or contrived) manner to specify such issues, but whether this is at the expense of good specification practice is open to debate.

We note here that the idea of developing a library of specification fragments that can be used to specify distributed systems has not been realised to the extent hoped. There were several reasons for this. Perhaps the main one is the lack of prescription in concepts. As discussed previously, ODP is not a standard to be used to develop a single system. It is a standard to be used to develop many systems, hence it cannot be overly prescriptive. As a result, the development of a library of specification fragments that could be put together to build distributed systems directly has not proven possible. We would argue that a more powerful form of re-usability has been developed here though, namely the re-usability of the approach. Some of the re-usable approaches we have developed here include:

- showing how qualitative timing issues can be represented in LOTOS and Z;
- showing how it is possible to use LOTOS to develop different forms of signature type checking;
- showing how interface references can be treated as first class citizens in LOTOS;
- showing how constraints on behaviours can be represented and checked in LOTOS and Z.

Each of these represents not a single fragment of text that can be used directly, but a source of ideas and approaches for modelling open distributed systems generally.

9.5 Areas of Future Work

The work presented here does not represent *the* final solution for specifying open distributed systems. It is possible to extend this work in numerous ways. For example, the non-functional aspects of systems might

be extended to include other constraints, *e.g.* location, reliability or availability constraints are just three possible ones.

The approach developed for allowing signature type checking to be achieved in LOTOS could well be extended. Instead of basing type checking on subsetting of operations, it is possible to have different approaches. For example, F-bounded polymorphism [39, 38] allows type relationships to be established depending on the operations required in a particular context. Thus if only a single operation is required in a given context then otherwise dissimilar type structures might be compatible if they share a common (compatible) operation. Similarly, it is quite possible to extend the approach given to modelling and using interface references in LOTOS so that the references them would not be static entities. Instead, interface references might be specified so that they are perpetually modified as operations occur, or behaviour more generally. For example, it is quite possible using the approach given to specify that the set of operations and constraints associated with an interface can change. This might be as direct as an increase in cost of using an interface, or the removal of an operation from the set of operations available at that interface. In a similar vein, it is possible to specify and reason about issues such as dynamic renegotiation of constraints or operations to be made available in bindings say, *e.g.* when more resources are added to the process.

The approach given in Act One might also be applied in a more practical environment. For example, the current ODP type management standardisation [116] is likely to require approaches that enable structural relationships between types to be established.

The approach given to modelling interactions between processes in LOTOS could also be extended. We have focussed primarily on interactions between pairs of objects. This need not necessarily be the case. It is possible for example to specify sets of interface references in parameter lists and model group forms of communication. Alternatively, interface references themselves might be modelled in Act One so that classes of interface references are specified. For example, a class might be based on the set of interface references with location X or supporting operation Y, say. With these models, different forms of communication possibilities can be achieved.

The approaches given here have shown how LOTOS and Z can be used in a stylised manner to specify open, object-based distributed systems. Ideally of course, stylised specifications should not be necessary; languages should have inherent features where specification styles are used to introduce these features. Extensions to Z to deal with objects-based systems are one step in the right direction. Other interesting works include composing Z and CSP [8, 72] to give Z an interaction semantics as opposed to the informal convention of inputs and outputs.

With regard to LOTOS, the current E-LOTOS standardisation [112] is certainly attempting to address many of the issues involved in specifying open, object-based distributed systems. These include extensions to LOTOS for dealing with timing issues explicitly. Hence in E-LOTOS timing issues are manifest in the semantics as well as the syntax of the language. In addition the language allows for the specification of prioritised actions.

The dynamicity of distributed systems is being addressed in E-LOTOS through providing a form of dynamic reconfigurability based on a mobile calculus [149]. E-LOTOS also provides features that resemble type checking as might be found in IDLs. This is achieved through gate typing. Whilst enabling a form of type checking to be achieved, gate typing represents only a partial solution to the checks required when composing systems together or replacing one system for another. Gate typing in effect allows actions to be type checked, whereas ideally it should be interfaces as a whole that are type checked. E-LOTOS is also providing features to deal with exceptions as might be found in IDLs.

Most specifiers are agreed that the Act One part of LOTOS specifications is both too verbose and does not have enough predefined and user-oriented data types. E-LOTOS has thus extended LOTOS with a new data typing language based on ML [84]. This contains a wider range of standard data types such as union types, recursive types, records and extensible records.

E-LOTOS also extends LOTOS with modules. The comparison of modules in LOTOS and distributed systems as collections of interworking objects is immediately apparent.

It is hoped that these areas of work have been influenced to some extent by the work undertaken in this thesis and the development of an architectural semantics for ODP more generally.

References

[1] R. Alderden. COOPER – the compositional construction of a canonical tester. In S. T. Vuong, editor, *Proc. Formal Description Techniques II*, pages 13–18. North-Holland, Amsterdam, Netherlands, 1990.

[2] A. J. Alencar and J. A. Goguen. OOZE: An object-oriented Z environment. In P. America, editor, *Proc. ECOOP'91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, 1991.

[3] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[4] ANSA, Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, UK. *The ANSA Reference Manual*, 1989.

[5] ANSA. The challenge of ODP. Technical Report TR.33.02, Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, UK, 1993.

[6] Apollo. *Network Computing Architecture (NCA) Protocol Specifications*. Apollo Computer Inc., 1989.

[7] R. Barden, S. Stepney, and D. Cooper. *Z in Practice.* BCS Practitioner Series. Prentice Hall, 1994.

[8] M. Benjamin. A message passing system. an example of combining CSP and Z. In *Z User Workshop, Workshops in Computing*. Springer-Verlag, 1989.

[9] J.K. Bennett. The design and implementation of distributed Smalltalk. In *OOPSLA'87 Proceedings, ACM SIGPLAN Notices*, volume 22, pages 318–330, October 1987.

[10] J. Bernabeu, Y. A. Khalidi, M. Ahamad, W.F. Appelbe, P. Dasgupta, R.J. LeBlanc, and U. Ramachandran. Clouds – a distributed object-oriented operating system: Architecture and kernel implementation. In *Proceedings of the 88'EUUG Conference*, October 1988.

[11] A.D Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, February 1984.

[12] D. Bjørner, C.A.R. Hoare, and H. Langmaack. *VDM'90: VDM and Z - Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[13] A. Black, N. Hutchinson, E. Jul, H. Levy, and L.Carter. Distribution and abstract data types in Emerald. *IEEE Transactions on Software Engineering*, SE-13 No. 1:65–76, 1987.

[14] Stewart Black. Objects and LOTOS. In S. T. Vuong, editor, *Proc. Formal Description Techniques II*. Elsevier Science Publishers, B.V. North Holland, December 1989.

[15] G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd. *Object-Oriented Languages, Systems and Applications*. Pitman Publishing, 1991.

[16] Gordon Blair, Lynne Blair, Howard Bowman, and Amanda Chetwynd. Formal support for the specification and construction of distributed multimedia systems (the Tempo project). Technical Report MPG-93-23, University of Lancaster, England, 1993.

[17] Gordon Blair, Amanda Chetwynd, Abderrahmane Lakas, and Neil Barnes. TEMPO: Specification and validation of quality of service in distributed multimedia systems. Technical Report ????, University of Lancaster, England, 1996.

[18] Gordon S. Blair and Rodger Lea. The impact of distribution on support for object-oriented software development. Technical Report MPG-93-25, University of Lancaster, England, 1993.

[19] Lynne Blair. *The Formal Specification and Verification of Distributed Multimedia Systems.* PhD thesis, Department of Computing Science, Lancaster University, 1994.

[20] Kees Bogaards. *A Methodology for the Architectural Design of Open Distributed Systems.* PhD thesis, University of Twente, NL, 1990.

[21] Paul Böhm, Jan de Meer, and Peter Schoo. Perlon persistency checker for data type definitions. In Ed Brinksma, Giuseppe Scollo, and Christopher A. Vissers, editors, *Proc. Protocol Specification, Testing and Verification VIII*, pages 285–302, 1989.

[22] Eerke A. Boiten, John Derrick, Howard Bowman, and Maarten W. A. Steen. Consistency and refinement for partial specification in Z. In M.-C. Gaudel and James Woodcock, editors, *Proc. Formal Methods Europe '96*, volume 1051 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag, March 1996.

[23] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, January 1987.

[24] Tommaso Bolognesi and M. Caneve. SQUIGGLES: A tool for the analysis of LOTOS specifications. In Kenneth J. Turner, editor, *Proc. Formal Description Techniques I*, pages 201–216. North-Holland, Amsterdam, Netherlands, 1989.

[25] Tommaso Bolognesi and F. Lucidi. LOTOS-like process algebras with urgent or timed interactions. In Kenneth R. Parker and Gordon A. Rose, editors, *Proc. Formal Description Techniques IV*, pages 249–264. North-Holland, Amsterdam, Netherlands, November 1991.

[26] G. Booch. *Software Engineering with Ada*. Benjamin Cummings, 1983.

[27] G. Booch. *Object-Oriented Design with Applications*. Benjamin-Cummins, second edition, 1991.

[28] J. P. Bowen and J. A. Hall, editors. *Z User Workshop, Cambridge 1994*, Workshops in Computing. Springer-Verlag, 1994.

[29] J.P. Bowen and M.G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):11–19, July 1995.

[30] J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods. *IEEE Software*, 28(4):56–63, April 1995.

[31] Howard Bowman, Eerke A. Boiten, John Derrick, and Maarten W. A. Steen. Viewpoint consistency in ODP, a general interpretation. In Elie Najm and Jean-Bernard Stefani, editors, *Formal Methods for Open Object-based Distributed Systems*, Paris, France, March 1996.

[32] Howard Bowman, John Derrick, Peter Linington, and Maarten W. A. Steen. Cross-viewpoint consistency in Open Distributed Processing. *Software Engineering Journal*, pages 44–57, January 1996.

[33] Rolv Bræk and Øystein Haugen. *Engineering Real Time Systems.* BCS Practitioner Series. Prentice Hall, 1993.

[34] P. T. Breuer and J. P. Bowen. Towards correct executable semantics for Z. In Bowen and Hall [28], pages 185–209.

[35] E. Brinksma and G. Scollo. Formal notions of implementation and conformance in LOTOS. Technical Report INF-86-13, Dept. of Informatics, Twente University of Technology, Enschede, NL, December 1986.

[36] K. Brockschmidt. *Inside OLE 2*. Microsoft Press, second edition, 1995.

[37] D. Brownbridge. Using Z to develop a CASE toolset. In Nicholls [151], pages 142–149.

[38] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *Communications of the ACM*, pages 457–4670, 1989.

[39] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. *ACM Computing Surveys*, 9:273–280, 1989.

[40] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[41] D.A. Carrington, D. Duke, R. Duke, P. King, G.A. Rose, and G. Smith. Object-Z: an object-oriented extension to Z. In Son Vuong, editor, *2nd International Conference on Formal Description Techniques (FORTE'89)*, pages 313–341. Elsevier Science Publishers, B.V. North Holland, 1989.

[42] T. Y. Cheung, Y. C. Ye, X. Ye, and G. Q. Wang. UO-Lotos: A syntax/system for representing, editing and translating graphical LOTOS. In S. T. Vuong, editor, *Proc. Formal Description Techniques II*, pages 31–36. North-Holland, Amsterdam, Netherlands, 1989.

[43] R. G. Clark and A. M. D. Moreira. Behavioural inheritance in ROOA. In *4th IS-CORE Workshop*, Amsterdam, The Netherlands, 1994.

[44] R.G. Clark. LOTOS design-oriented specifications in the object-based style. Technical Report 84, Department of Computing Science and Mathematics, University of Stirling, April 1992.

[45] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press, Englewood Cliffs, NJ USA, 2nd edition, 1991.

[46] P. Coad and E. Yourdon. *Object-Oriented Design*. Yourdon Press, Englewood Cliffs, NJ USA, 2nd edition, 1991.

[47] A. Coombes. An interval logic for modelling time in Z. Technical report, Department of Computing Science, University of York, 1990.

[48] Xerox Corporation. *Courier: the Remote Procedure Call Protocol*. Xerox System Integration Standard, XSIS-038112, Xerox Corporation, December 1991.

[49] Jean-Pierre Courtiat and Djamel E. Saïdouni. Action refinement in LOTOS. In André A. S. Danthine, Guy Leduc, and Pierre Wolper, editors, *Proc. Protocol Specification, Testing and Verification XIII*, pages 341–354. North-Holland, Amsterdam, Netherlands, May 1993.

[50] D. Craigen, S. L. Gerhart, and T. J. Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, 1995.

[51] J. Crowcroft. *Open Distributed Systems.* University College London Press, Gower Street, London, 1996.

[52] E. Cusack. Inheritance in object-oriented Z. In P. America, editor, *Proc. ECOOP'91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 167–179. Springer-Verlag, 1991.

[53] E. Cusack. Object-oriented modelling in Z for open distributed systems. In J. de Meer, editor, *Proc. International Workshop on ODP*. Elsevier Science Publishers (North-Holland), 1992.

[54] E. Cusack and M. Lai. Object-oriented specification in LOTOS and Z (or my cat really is object-oriented!). In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *REX/FOOL School/Workshop on Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 179–202. Springer-Verlag, 1990.

[55] E. Cusack and H. B. Zadeh. ZEST — Z Extended with Structuring. In Stepney et al. [186], pages 100–113.

[56] E. Cusack and H. B. Zadeh. ZEST — Z Extended with Structuring. In Lano and Haughton [123].

[57] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):28–79, March 1988.

[58] André A.S. Danthine. *The OSI95 Transport Service with Multimedia Support*. Springer-Verlag, 1994.

[59] J. de Meer, R. Roth, and S. Vuong. Introduction to algebraic specifications based on the language act one. In *Computer Networks and ISDN Systems*, volume 23, pages 363–392. Elsevier Science Publishers (North-Holland), 1992.

[60] N. Delisle and D. Garlan. Formally specifying electronic instruments. *ACM SIGSOFT Eng. Notes*, 14:242–248, 1989.

[61] J.S. Dong and R. Duke. An object-oriented approach to the formal specification of ODP trader. In *Proc. IFIP TC6/WG6.1 International Conference on Open Distributed Processing*, pages 341–352, September 1993.

[62] J.S. Dong, R. Duke, and G. A. Rose. An object-oriented approach to the semantics of programming languages. In *Proc. 17th Australian Computer Science Conference (ACSC-17)*, pages 767–775, January 1994.

[63] D. Duke and R. Duke. Towards a semantics for Object-Z. In *VDM'90: VDM and Z - Formal Methods in Software Development* [12], pages 244–261.

[64] H. Eertink. Executing LOTOS specifications: The SMILE tool. In *Third LotoSphere Workshop and Seminar*, September 1992.

[65] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[66] Jonathan Erickson. Distributed computing. *Dr. Dobb's Journal*, 20(2), February 1995.

[67] Andrew Evans. OO Oversold. *Information and Software Technology*, 36(1):35–42, 1994.

[68] K. Farooqui and L. Logrippo. Viewpoint transformation. In *Proc. IFIP TC6/WG6.1 International Conference on Open Distributed Processing*, pages 352–363, September 1993.

[69] J. C. Fernandez, Hubert Garavel, L. Mounier, A. Rasse, and C. Rodriguez. A toolbox for the verification of LOTOS programs. In *Proc. 14th International Conference on Software Engineering and its Applications*, pages 246–259, May 1992.

[70] C. Fidge. Specification and verification of real-time behaviour using Z and RTL. In *Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 393–410. Springer-Verlag, 1992.

[71] Colin Fidge. A comparative introduction to CSP, CCS and LOTOS. Technical Report 93-24, Department of Computer Science, University of Queensland, Brisbane, Australia, April 1994.

[72] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In *Second Conference on Formal Methods for Open Object-based Distributed Systems*. Chapman and Hall, London, UK, 1997.

[73] J. Fischer, A. Prinz, and A. Vogel. Different FDTs confronted with different ODP-Viewpoints of the Trader. In *FME'93: Industrial Strength, Formal Methods, First International Symposium of Formal Methods Europe*, pages 332–349. Lecture Notes in Computer Science, 1993.

[74] M. Flynn, T. Hoverd, and D. Brazier. Formaliser – an interactive support tool for Z. In Nicholls [151], pages 128–141.

[75] Iain Fogg, Brian Hicks, Andrew Lister, Tim Mansfield, and Kerry Raymond. A comparison of LOTOS and Z for specifying distributed systems. *Australian Computer Science Community*, 12(1):88–96, February 1990.

[76] N. E. Fuchs. Specifications are (preferably) executable. *IEE/BCS Software Engineering Journal*, 7(5):323–334, September 1992.

[77] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 707 of *Lecture Notes in Computer Science*, pages 407–431. Springer-Verlag, July 1993.

[78] Hubert Garavel. On the introduction of gate typing in E-LOTOS. In Piotr Dembinski and Marek Sredniawa, editors, *Proceedings of the 15th IFIP International Workshop on Protocol Specification, Testing and Verification*, pages 273–288. North-Holland, Amsterdam, Netherlands, June 1995.

[79] Hubert Garavel and Joseph Sifakis. Compilation and verification of LOTOS specifications. In Luigi M. S. Logrippo, Robert L. Probert, and Hassan Ural, editors, *Proc. Protocol Specification, Testing and Verification X*, Amsterdam, Netherlands, June 1990. North-Holland.

[80] V. Gay, P. Leydekkers, and R.H. in 't Veld. Specification of multi-party audio and video interaction based on the Reference Model of Open Distributed Processing. *Computer Networks and ISDN Systems*, 27:1247–1262, 1995.

[81] J. Paul Gibson. *Formal Object-Oriented Development of Software Systems using LOTOS*. PhD thesis, Department of Computing Science and Mathematics, University of Stirling, December 1993.

[82] J. A. Goguen, C. Kirchner, A. Megrelis, J. Meseguer, and T. Winkler. An introduction to OBJ3. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, 1987.

[83] A. Goldberg and D. Robson. *Smalltalk 80: The Language*. Addison-Wesley Publishing Company, Reading, MA, USA, 1989.

[84] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *ML*. Edinburgh LCF: Springer Verlag, 1979.

[85] R. Gotzhein. The formal definition of architectural concepts: "Interaction Points". In Son Vuong, editor, *2nd International Conference on Formal Description Techniques (FORTE'89)*, pages 1263–1304. ESP, 1989.

[86] R. Gotzhein. Towards a basic reference model of Open Distributed Processing. *Computer Networks and ISDN Systems*, 27:1263–1304, 1995.

[87] Object Management Group. *The Common Object Request Broker Architecture and Specification: Revision 2.0*. Object Management Group, Inc., Framingham, MA, July 1995.

[88] Object Management Group. *CORBA Facilities*. Object Management Group, Inc., Framingham, MA, March 1995.

[89] Object Management Group. *CORBA Services.* Object Management Group, Inc., Framingham, MA, March 1995.

[90] Object Management Group. *Object Management Architecture Guide*. Object Management Group, Inc., Framingham, MA, November 1992.

[91] J. A. Hall. Specifying and interpreting class hierarchies in Z. In Bowen and Hall [28], pages 120–138.

[92] J.A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[93] J.A. Hall. Using Z as a specification calculus for object-oriented systems. In *VDM'90: VDM and Z - Formal Methods in Software Development* [12], pages 290–318.

[94] I.J Hayes and C.B. Jones. Specifications are not (necessarily) executable. *IEE/BCS Software Engineering Journal*, 4(6):320–338, November 1989.

[95] Andrew J. Herbert. Personal communication, January 1997.

[96] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, NJ USA, 1985.

[97] I. Houston and M. Josephs. A formal description of the OMG's Core Object Model and the meaning of compatible extension. *Computer Standards & Interfaces*, 17:553–558, 1995.

[98] R. L. Ibrahim, J.A. Ogden, and S.A. Williams. Should concurrency be specified? In *Specification and Verification of Concurrent Systems* [163], pages 246–271.

[99] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Abstract Syntax Notation One (ASN.1)*. ISO/IEC 8824. International Organization for Standardization and International Electrotechnical Committee, Geneva, Switzerland, 1987.

[100] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Estelle – A Formal Description Technique Based on an Extended State Transition Model*. ISO/IEC 9074. International Organization for Standardization, Geneva, Switzerland, 1989.

[101] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 8807. International Organization for Standardization, Geneva, Switzerland, 1989.

[102] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Conventions for the Definition of OSI Services.* ISO/IEC TR 10731. International Organization for Standardization, Geneva, Switzerland, 1992.

[103] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Basic Reference Model*. ISO/IEC 7498. International Organization for Standardization, Geneva, Switzerland, 1994.

[104] ISO/IEC. *Quality of Service: Basic Framework.* ISO/IEC JTC1/SC21 N9309. ISO/IEC ITU-T, Geneva, Switzerland, 1995.

[105] ISO/IEC. *Quality of Service: Methods and Mechanisms*. ISO/IEC JTC1/SC21 N9310. ISO/IEC ITU-T, Geneva, Switzerland, 1995.

[106] ISO/IEC. *Z Notation version 1.2.* ISO/IEC JTC1/SC22/WG19 CD 13568. ISO/IEC, Geneva, Switzerland, August 1995.

[107] ISO/IEC. *Basic Reference Model of ODP – Part 1: Overview and Guide to Use of the Reference Model*. Draft International Standard 10746-1, Draft ITU-T Recommendation X.901. ISO/IEC ITU-T, Geneva, Switzerland, 1996.

[108] ISO/IEC. *Basic Reference Model of ODP – Part 2: Foundations*. International Standard 10746-2, ITU-T X.902. ISO/IEC ITU-T, Geneva, Switzerland, 1996.

[109] ISO/IEC. *Basic Reference Model of ODP – Part 3: Architecture*. International Standard 10746-3, ITU-T X.903. ISO/IEC ITU-T, Geneva, Switzerland, 1996.

[110] ISO/IEC. *Basic Reference Model of ODP – Part 4: Architectural Semantics*. Draft International Standard 10746-4, Draft ITU-T Recommendation X.904. ISO/IEC ITU-T, Geneva, Switzerland, 1996.

[111] ISO/IEC. *Basic Reference Model of ODP – Part 4.1: Architectural Semantics Amendment.* ISO/IEC JTC1/SC21 Working Document N9818. ISO/IEC ITU-T, Geneva, Switzerland, 1996.

[112] ISO/IEC. *Extended-LOTOS (E-LOTOS)*. ISO/IEC JTC1/SC21 WG7 Working Draft (WI1.21.20.2.3). ISO/IEC ITU-T, Geneva, Switzerland, January 1997. official ISO/IEC number yet to be assigned.

[113] ISO/IEC. *Open Distributed Processing — Interface References and Binding*. ISO/IEC JTC1/SC21 Commitee Draft N10430. ISO/IEC ITU-T, Geneva, Switzerland, May 1996.

[114] ISO/IEC. *Open Distributed Processing — ODP Enterprise Standard*. ISO/IEC JTC1/SC21 Working Document N10387. ISO/IEC ITU-T, Geneva, Switzerland, May 1996.

[115] ISO/IEC. *Open Distributed Processing — Trader*. ISO/IEC JTC1/SC21 Draft International Standard 13235. ISO/IEC ITU-T, Geneva, Switzerland, May 1996.

[116] ISO/IEC. *Open Distributed Processing — Type Repository Function*. ISO/IEC JTC1/SC21 Working Document N10389. ISO/IEC ITU-T, Geneva, Switzerland, May 1996.

[117] ISO/IEC. *PREMO — Presentation Environment for Multimedia Objects*. ISO/IEC JTC1/SC24 Commitee Draft N1592. ISO/IEC ITU-T, Geneva, Switzerland, September 1995.

[118] ITU-T. *International Consultative Committee on Telegraphy and Telephony – SDL – Specification and Description Language*. CCITT Z.100. International Telecommunications Union, Geneva, Switzerland, 1992.

[119] F. Jahanian and A. Mok. Safety analysis of timing properties in real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium, Santa Monica*, pages 319–328. IEEE Computer Society Press, 1989.

[120] Haim Kilov and James Ross. *Information Modelling: An Object-Oriented Approach*. Prentice Hall, Englewood Cliffs, NJ USA, 1993.

[121] P. King. A formal specification of signalling system number 7 link layer. Technical Report TR-101, University of Queensland, Key Centre for Software Technology, 1989.

[122] K. C. Lano and H. P. Haughton. *The Z^{++} Manual*. Lloyd's Register of Shipping, 29 Wellesley Road, Croydon CRO 2AJ, UK, 1992.

[123] K. C. Lano and H. P. Haughton, editors. *Object-Oriented Specification Case Studies*. Object-Oriented Series. Prentice Hall International, 1993.

[124] Christina Lau. *Object-Oriented Programming Using SOM and DSOM*. Van Nostrand Reinhold, New York, 1994.

[125] G. Leduc. A framework based on implementation relations for implementing LOTOS specifications. *Computer Networks and ISDN Systems*, 25(1):23–41, August 1992.

[126] Guy Leduc. An upward compatible timed extension to LOTOS. In Kenneth R. Parker and Gordon A. Rose, editors, *Proc. Formal Description Techniques IV*, pages 223–238. North-Holland, Amsterdam, Netherlands, 1991.

[127] Guy Leduc and Luc Léonard. A timed LOTOS supporting a dense time domain and including new timed operators. In Michel Diaz and Roland Groz, editors, *Proc. Formal Description Techniques V*, pages 87–102. North-Holland, Amsterdam, Netherlands, October 1992.

[128] L. Leonard, G. Leduc, and A. Danthine. The tick-tock case study for the assessment of timed FDTs. In *The OSI95 Transport Service with Multimedia Support* [58], pages 338–352.

[129] H. Leopold, G. Coulson, K. Frimpong-Ansah, D. Hutchison, and N. Singer. The evolving relationship between OSI and ODP in the new communications environment. Technical Report MPG-93-16, University of Lancaster, England, 1993.

[130] P. F. Linington. *Why objects have more than one interface*. ISO/IEC JTC1 SC21/WG7 Draft Answer to Q7/5. International Organization for Standardization, 1995.

[131] Luigi M. S. Logrippo, Mohammed Faci, and Mazen Haj-Hussein. An introduction to LOTOS: Learning by examples. *Computer Networks and ISDN Systems*, 23:325–342, 1992.

[132] E. Madelaine and D. Vergamini. AUTO: A verification tool for distributed system using reduction of finite state automata networks. In S. T. Vuong, editor, *Proc. Formal Description Techniques II*, pages 79–84, Amsterdam, Netherlands, December 1989. North-Holland.

[133] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented DBMS. In *Proceedings of the OOPSLA'86 Conference*, pages 472–482, 1986.

[134] Jose A. Mañas and Tomas de Miguel Moro. From LOTOS to C. In Kenneth J. Turner, editor, *Proc. Formal Description Techniques I*, pages 79–84. North-Holland, Amsterdam, Netherlands, 1989.

[135] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, Berlin, Germany, 1992.

[136] Ashley McClenaghan. *Distributed Systems: Architecture-Driven Specification using Extended LOTOS*. PhD thesis, Department of Computing Science and Mathematics, University of Stirling, September 1993.

[137] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall International Series in Computing Science: C.A.R. Hoare Series Editor. Prentice Hall, Englewood Cliffs, NJ USA, 1988.

[138] C. Miguel, A. Fernandez, and L. Vidaller. Extending LOTOS towards performance evaluation. In Michel Diaz and Roland Groz, editors, *Proc. Formal Description Techniques V*, pages 103–118. North-Holland, Amsterdam, Netherlands, October 1992.

[139] C. Miguel, L. Vidaller, and A. Fernández. Extended LOTOS. In *The OSI95 Transport Service with Multimedia Support* [58], pages 312–337.

[140] Erich Mikk. Compilation of Z Specifications into C for Automatic Test Result Evaluation. In Jonathan P. Bowen and Michael G. Hinchey, editors, *Proceedings of the 9th International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*, pages 167–180. Springer-Verlag, September 1995.

[141] Robin Milner. *A Calculus of Communicating Systems*, volume 92. Lecture Notes in Computer Science, 1980.

[142] Robin Milner. *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, NJ USA, 1989.

[143] Robin Milner. The polyadic pi-calculus: A tutorial. In F. L. Hamer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, 1993.

[144] A. Moreira and R.G. Clark. Chapter 3 in LOTOS in the object-oriented analysis process. In *Formal Methods and Object Technology*. Springer-Verlag, Berlin, Germany, 1996.

[145] A. M. D. Moreira and R. G. Clark. Combining object-oriented analysis and formal description techniques. In M. Tokoro and R. Pareschi, editors, *ECOOP'94*, volume 821 of *Lecture Notes in Computer Science*, pages 344–364. Springer-Verlag, 1994.

[146] A. M. D. Moreira and R. G. Clark. Rigorous object-oriented analysis. In *International Symposium on Object-Oriented Methodologies and Systems (ISOOMS)*, Palermo, Italy, 1994.

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

[148] E. Najm and J.B. Stefani. A formal semantics for the ODP computational model. *Computer Networks and ISDN Systems*, 27:1305–1329, 1995.

[149] Elie Najm and Jean-Bernard Stefani. Dynamic configuration in LOTOS. In Kenneth R. Parker and Gordon A. Rose, editors, *Proc. Formal Description Techniques IV*, pages 201–216. North-Holland, Amsterdam, Netherlands, November 1991.

[150] B.J. Nelson. *Remote Procedure Call*. PhD thesis, Department of Computing Science, Carnegie Mellon University, 1981.

[151] J. E. Nicholls, editor. *Z User Workshop, Oxford 1989*, Workshops in Computing. Springer-Verlag, 1990.

[152] J.R. Nicol, C.T. Wilkes, and F.A. Manola. Object-orientation in heterogeneous distributed computing systems. In *IEEE Computer*, volume 26(6), pages 57–67. IEEE Computer Press, June 1993.

[153] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Proceedings of the 3rd Workshop on Computer Aided Verification (CAV'91)*, pages 526–548. Springer-Verlag, Aalborg, Denmark, July 1991.

[154] S. O'Brien. *Jigsaw*. Tutorial presented at the 9th International Conference of Z Users, Limerick Ireland, 1996.

[155] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley and Sons, inc., New York, USA, 1996.

[156] A. Parhar. *TINA Object Definition Language Manual version 2.3*. Internal Document Number TR-NM-.002-2.3-96. TINA-C, May 1996.

[157] C. E. Parker. Z tools catalogue. ZIP project report ZIP/BAe/90/020, British Aerospace, Software Technology Department, Warton PR4 1AX, UK, May 1991.

[158] G.I. Parkin and S. Austin. Overview: Survey of formal methods in industry. In *Proceedings of Formal Description Techniques VI*, pages 189–204. North Holland, Amsterdam, Netherlands, 1994.

[159] P.F. Pinto and P.F. Linington. A language for the specification of interactive distributed multimedia applications. In *Proc. IFIP TC6/WG6.1 International Conference on Open Distributed Processing*, pages 217–234, September 1993.

[160] A. Pnueli. The temporal logic of programs. In *Proceedings of the 12th International Colloquium on Automata, Languages and Programming*, volume 194 of *Lecture Notes in Computer Science*, pages 15–32, Berlin, Germany, 1985. Springer-Verlag.

[161] Ben Potter, Jane Sinclair, and David Till, editors. *An Introduction to Formal Specification and Z*. Prentice Hall International, Englewood Cliffs, New Jersey, USA, 1991.

[162] Juan Quemada, Arturo Azcorra, and David de Frutos. TIC: A timed calculus for LOTOS. In S. T. Vuong, editor, *Proc. Formal Description Techniques II*, pages 195–209. North-Holland, Amsterdam, Netherlands, 1990.

[163] C. Rattray. *Specification and Verification of Concurrent Systems*. Workshops in Computing. Springer-Verlag, 1990.

[164] S. Rudkin. Inheritance in LOTOS. In *Fourth International Conference on Formal Description Techniques*. Elsevier Science Publishers, B.V. North Holland, November 1991.

[165] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, NJ USA, 1991.

[166] A. Sampaio and S. Meira. Modular extensions to Z. In *VDM'90: VDM and Z - Formal Methods in Software Development* [12], pages 211–232.

[167] Ina K. Schieferdecker. *Performance-Oriented Specification of Communication Protocols and Verification of Deterministic Bounds of their QoS Characteristics*. PhD thesis, Technical University of Berlin, Germany, 1994.

[168] Jeroen Schot. *The Role of Architectural Semantics in the formal approach of Distributed Systems Design*. PhD thesis, University of Twente, NL, February 1992.

[169] S.A. Schuman, D.H. Pitt, and P.J. Byers. Object-oriented process specification. In *Specification and Verification of Concurrent Systems* [163], pages 21–70.

[170] J. Shirley. *Guide to Writing DCE Applications*. O'Reilly and Associates, Inc., Sebastopol CA, USA, 1993.

[171] Jon Siegel. *CORBA Fundamentals and Programming*. John Wiley and Sons, Inc., New York, USA, 1996.

[172] Richard O. Sinnott. Type checking in distributed systems: an overview of current approaches. First Milestone for GMD-Fokus, December 1996.

[173] R.O. Sinnott. The formally specifying of electronic components in LOTOS. Master's thesis, Department of Computing Science and Mathematics, University of Stirling, 1994.

[174] R.O. Sinnott and K. J. Turner. DILL: Specifying digital logic in LOTOS. In Ü. Uyar R.L. Tenney, P.D. Amer, editor, *Proceedings of Formal Description Techniques VI*, pages 71–86. Elsevier Science Publishers, B.V. North Holland, 1994.

[175] R.O. Sinnott and K. J. Turner. Applying the architectural semantics of ODP to develop a trader specification. *Computer Networks and ISDN Systems: Special Edition on Specification Architecture*, March 1997.

[176] R.O. Sinnott and K.J. Turner. Modelling ODP viewpoints. In H. Kilov and W. Harvey, editors, *Proceedings of OOPSLA'94 Workshop on Precise Behavioural Semantics*, Portland, Oregon, October 1994.

[177] R.O. Sinnott and K.J. Turner. Specifying multimedia binding objects in Z. In O. Spaniol, C. Popien, and B. Meyer, editors, *Trends in Distributed Systems: CORBA and Beyond*, Lecture Notes in Computer Science, October 1996.

[178] R.O. Sinnott and K.J. Turner. Specifying ODP Computational Objects in Z. In Elie Najm and Jean-Bernard Stefani, editors, *Proceedings of Formal Methods for Open Object-based Distributed Systems*, Paris, France, March 1996.

[179] R.O. Sinnott and K.J. Turner. Type checking in distributed systems: a complete model and its Z specification. In J. Rolia, editor, *International Conference on Open Distributed Processing (ICODP) and Distributed Platforms (ICDP)*, May 1997.

[180] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, January 1988.

[181] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International Series in Computing Science: C.A.R. Hoare Series Editor. Prentice Hall, Englewood Cliffs, NJ USA, second edition, 1992.

[182] J.M. Spivey. *The Fuzz Manual*. Computing Science Consultancy, 1993. Second Printing.

[183] Maarten W. A. Steen, Howard Bowman, and John Derrick. Composition of LOTOS specifications. In Piotr Dembinski and Marek Sredniawa, editors, *Proceedings of the 15th International Symposium on Protocol Specification, Testing and Verification (PSTV XIII)*, pages 87–102. Chapman and Hall, London, UK, June 1995.

[184] J.B. Stefani. Computational aspects of QoS in an object-based distributed systems architecture. In *3rd International Workshop on Responsive Computer Systems*, Lincoln, NH, USA, September 1993.

[185] Pete Steggles and Jason Hulance. Z tools survey. Technical report, Imperial Software Technology, Cambridge, UK, June 1994.

[186] S. Stepney, R. Barden, and D. Cooper, editors. *Object-Orientation in Z*. Workshops in Computing. Series Edited by Prof. C.J. van Rijsbergen. Springer-Verlag, 1992.

[187] W.J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[188] Phil Stocks, Kerry Raymond, David Carrington, and Andrew Lister. Modelling open distributed systems in Z. *Computer Communications*, 15(2):103–113, 1992.

[189] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, USA, 1991.

[190] A. S. Tanenbaum and R. van Renesse. A critique of the remote procedure call paradigm. *Research into Networks and Distributed Applications*, 1988.

[191] H.C. Tijms. *Stochastic Modelling and Analysis: a Computational Approach*. John Wiley and Sons, 1986.

[192] Alistair J. Tocher. Towards a theory of objects. Technical Report RC.066.02, Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, UK, May 1990.

[193] I. Toyn and J. A. McDermid. CADiZ: An architecture for Z tools and its implementation. Technical document, Computer Science Department, University of York, York YO1 5DD, UK, November 1993.

[194] K. J. Turner. *Using Formal Description Techniques: An Introduction to Estelle, LOTOS and SDL*. John Wiley and Sons, 1993.

[195] K. J. Turner. Relating architecture and specification. *Computer Networks and ISDN Systems: Special Edition on Specification Architecture*, March 1997.

[196] K. J. Turner. Specification architecture in a communications context. *Computer Networks and ISDN Systems: Special Edition on Specification Architecture*, March 1997.

[197] Kenneth J. Turner. LOTOS – A practical Formal Description Technique for OSI. In *International Open Systems 87*, volume 1, pages 265–279. Online Publications, London, March 1987.

[198] Ray Vald'es. Interoperable objects. *Dr. Dobb's Special Report*, 19(6), Winter 1994/5.

[199] S. Valentine. The programming language Z^{--} . *Information and Software Technology*, 37(5):293–301, May 1995.

[200] Peter H. J. van Eijk. LOTOS tools based on the Cornell Synthesiser Generator. In Ed Brinksma, Giuseppe Scollo, and Christopher A. Vissers, editors, *Proc. Protocol Specification, Testing and Verification VIII*, pages 43–52. North-Holland, Amsterdam, Netherlands, June 1989.

[201] Peter H. J. van Eijk. The LOTOSPHERE integrated tool environment LITE. In Kenneth R. Parker and Gordon A. Rose, editors, *Proc. Formal Description Techniques IV*, pages 471–474. North-Holland, Amsterdam, Netherlands, November 1991.

[202] Wilfried H. P. van Hulzen, Paul A. J. Tilanus, and Han Zuidweg. LOTOS extended with clocks. In S. T. Vuong, editor, *Proc. Formal Description Techniques II*, pages 179–193. North-Holland, Amsterdam, Netherlands, 1990.

[203] A. Vogel. *Entwurf, Realisierung und Test von ODP-Systemen auf der Grundlage formaler Beschreibungstechniken*. PhD thesis, Humboldt-Universität zu Berlin, 1993. In German.

[204] W. Rosenberry, D. Kenney and G. Fisher. *Understanding DCE*. O'Reilly and Associates, Inc., Sebastopol CA, USA, 1993.

[205] P. Wegner. The object-oriented classification paradigm. In P. Wegner and B. Shriver, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[206] S. Weinberg. *The first three minutes*. Bantam Press, 1995.

[207] Clazien D. Wezeman. The CO-OP method for compositional derivation of conformance testers. In Ed Brinksma, Giuseppe Scollo, and Christopher A. Vissers, editors, *Proc. Protocol Specification, Testing and Verification VIII*. North-Holland, Amsterdam, Netherlands, June 1989.

[208] J. E. White. A high level framework for network-based resource sharing. *Proceedings of the National Computer Conference*, June 1976.

[209] P. J. Whysall and J. A. McDermid. An approach to object-oriented specification using Z. In J. E. Nicholls, editor, *Z User Workshop, Oxford 1990*, Workshops in Computing, pages 193–215. Springer-Verlag, 1991.

[210] P. J. Whysall and J. A. McDermid. Object-oriented specification and refinement. In J. M. Morris and R. C. Shaw, editors, *4th Refinement Workshop*, Workshops in Computing, pages 151–184. Springer-Verlag, 1991.

[211] J-P. Wu and S. Chanson. Translation from LOTOS and Estelle specifications to extended transition system and its verification. In Son Vuong, editor, *2nd International Conference on Formal Description Techniques (FORTE'89)*, pages 533–549. Elsevier Science Publishers, B.V. North Holland, 1989.

Appendix A

Specification of a Type Management System in LOTOS

This appendix shows the specification of a type management system in LOTOS that enables structural equivalences between types to be established. This specification does not attempt to deal with higher order type systems where issues of contravariance and covariance have to be addressed. However, the approach given could well be extended to deal with such systems.

```
(* Specification of a first order type management system in LOTOS *)

specification type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r]:noexit

library
      Boolean, NaturalNumber, Set
endlib

(* Names used to identify types *)

type NameSort is Boolean, NaturalNumber
     sorts NameSort
     opns NULL_SORT,typename1,typename2,typename3,typename,
          sort1, sort2, sort3, sort4:                      -> NameSort
          _ eq _ : NameSort, NameSort                      -> Bool
          _ ne _ : NameSort, NameSort                      -> Bool
          Ord    : NameSort                                -> Nat
     eqns forall n1, n2: NameSort
          ofsort Nat
            Ord(sort1) = 0;
            Ord(sort2) = succ(0);
            Ord(sort3) = succ(succ(0));
            Ord(sort4) = succ(succ(succ(0)));
            Ord(typename1) = 0;
            Ord(typename2) = succ(0);
            Ord(typename3) = succ(succ(0));
            Ord(typename4) = succ(succ(succ(0)));
            Ord(NULL_SORT) = succ(succ(succ(succ(0))));
          ofsort Bool
            n1 eq n2   = Ord(n1) eq Ord(n2);
            n1 ne n2   = Ord(n1) ne Ord(n2);
endtype (* Name *)

(* Sort lists used as input and output parameters to operations *)

type Sort_List is NameSort
     sorts Sort_List
     opns <>: -> Sort_List
          add_sort: NameSort, Sort_List  -> Sort_List
```

```
          del_sort: NameSort, Sort_List  -> Sort_List
          is_in   : NameSort, Sort_List  -> Bool
          is_eq   : Sort_List, Sort_List -> Bool
          _conc_  : Sort_List, Sort_List -> Sort_List
          head    : Sort_List            -> NameSort
          tail    : Sort_List            -> Sort_List
     eqns forall sn,sn1: NameSort, sl1,sl2: Sort_List
          ofsort NameSort
            head(<>)              = NULL_SORT;
            head(add_sort(sn,sl1)) = sn;
          ofsort Sort_List
            tail(<>)              = <>;
            tail(add_sort(sn,sl1)) = sl1;
            del_sort(sn,add_sort(sn,sl1)) = sl1;
            del_sort(sn,<>)       = <>;
            <> conc <>            = <>;
            sl1 conc <>           = sl1;
            sl1 conc add_sort(sn,sl2) = add_sort(sn,sl1) conc sl2;
          ofsort Bool
            (head(sl1) eq NULL_SORT)=> is_in(sn,sl1) = false;
            (head(sl1) ne NULL_SORT)=> is_in(sn,sl1) = (sn eq head(sl1)) or
                                                       (is_in(sn,tail(sl1)));
            is_eq(<>,<>)     = true;
            (head(sl1) ne NULL_SORT)=> is_eq(sl1,<>) = false;
            (head(sl2) ne NULL_SORT)=> is_eq(<>,sl2) = false;
            ((head(sl1) ne NULL_SORT) and (head(sl2) ne NULL_SORT))=>
                              is_eq(sl1,sl2) = (head(sl1) eq head(sl2))
                                               and is_eq(tail(sl1),tail(sl2));
endtype (* Sort_List *)

(* Names of operations *)

type OpName is Boolean, NaturalNumber
     sorts OpName
     opns opn1, opn2, opn3, opn4: -> OpName
          Ord : OpName -> Nat
          _eq_: OpName, OpName -> Bool
          _ne_: OpName, OpName -> Bool
     eqns forall op1, op2: OpName
          ofsort Nat
            Ord(opn1) = 0;
            Ord(opn2) = succ(0);
            Ord(opn3) = succ(succ(0));
            Ord(opn4) = succ(succ(succ(0)));
          ofsort Bool
            op1 eq op2   = Ord(op1) eq Ord(op2);
            op1 ne op2   = Ord(op1) ne Ord(op2);
endtype (* OpName *)

(* Operations consist of a name, a list of input sorts and a
   list of output sorts *)

type Op is OpName, Sort_List
     sorts Op
     opns make_op    : OpName, Sort_List, Sort_List -> Op
          get_name   : Op -> OpName
          get_inputs : Op -> Sort_List
          get_outputs: Op -> Sort_List
          _eq_       : Op, Op -> Bool
          _ne_       : Op, Op -> Bool
     eqns forall op1,op2: Op, opn1: OpName, sl1,sl2: Sort_List
          ofsort OpName
           get_name(make_op(opn1,sl1,sl2)) = opn1;
          ofsort Sort_List
           get_inputs(make_op(opn1,sl1,sl2)) = sl1;
           get_outputs(make_op(opn1,sl1,sl2)) = sl2;
          ofsort Bool
```

```
            op1 eq op2 = ((get_name(op1) eq get_name(op2)) and
                          (is_eq(get_inputs(op1),get_inputs(op2))) and
                          (is_eq(get_outputs(op1),get_outputs(op2))));
            op1 ne op2 = ((get_name(op1) ne get_name(op2)) or
                          (not(is_eq(get_inputs(op1),get_inputs(op2)))) or
                          (not(is_eq(get_outputs(op1),get_outputs(op2)))));
endtype (* Op *)

(* Generate sets of operations to enable data types to be built *)

type OpSet is Set actualizedby Op using
       sortnames OpSet for Set
                 Op for Element
                 Bool for FBool
endtype (* OpSet *)

(* Data types should allow names, sort lists, sets of
   operations to be established and subtyping to be checked *)

type DataType is OpSet
     sorts DataType
     opns NULL_DT, DT1, DT2, DT3, DT4: -> DataType
          is_a: NameSort -> DataType        (* get data type with given name *)
          IS_A: DataType -> NameSort        (* get name of data type *)
          sort: DataType -> Sort_List       (* return sort lists of data type *)
          ops : DataType -> OpSet           (* return operations on data type *)
          sub : DataType, DataType -> Bool  (* subtype checking operation *)

(* Build some data types for checking functionality of type manager *)

     eqns forall op1, op2: Op, opl1,opl2: OpSet,
                 adt1, adt2, adt3, adt4: DataType
          ofsort Sort_List
            sort(DT1) = add_sort(sort1,<>);
            sort(DT2) = add_sort(sort2,add_sort(sort1,<>));
            sort(DT3) = add_sort(sort3,add_sort(sort2,add_sort(sort1,<>)));
            sort(DT4) = add_sort(sort4,add_sort(sort3,add_sort(sort2,add_sort(sort1,<>))));
            sort(NULL_DT) = <>;
          ofsort OpSet
            ops(DT1) = Insert(make_op(opn1,<>,add_sort(sort1,<>)), {});
            ops(DT2) = Insert(make_op(opn1,<>,add_sort(sort1,<>)),
                              Insert(make_op(opn2,add_sort(sort1,<>),add_sort(sort2,<>)),{}));
            ops(DT3) = Insert(make_op(opn1,<>,add_sort(sort1,<>)),
                              Insert(make_op(opn2,add_sort(sort1,<>),add_sort(sort2,<>)),
                              Insert(make_op(opn3,add_sort(sort1,add_sort(sort2,<>)),
                                                  add_sort(sort3,<>)), {})));
            ops(DT4) = Insert(make_op(opn1,<>,add_sort(sort1,<>)),
                              Insert(make_op(opn2,add_sort(sort1,<>),add_sort(sort2,<>)),
                              Insert(make_op(opn3,add_sort(sort1,add_sort(sort2,<>)),
                                                  add_sort(sort3,<>)),
                              Insert(make_op(opn4,add_sort(sort1,add_sort(sort2,
                                        add_sort(sort3,<>))), add_sort(sort4,<>)), {})));
            ops(NULL_DT) = {};
          ofsort DataType
            is_a(typename1) = DT1;
            is_a(typename2) = DT2;
            is_a(typename3) = DT3;
            is_a(typename4) = DT4;
            is_a(NULL_SORT) = NULL_DT;
          ofsort NameSort
            IS_A(DT1) = typename1;
            IS_A(DT2) = typename2;
            IS_A(DT3) = typename3;
            IS_A(DT4) = typename4;
            IS_A(NULL_DT) = NULL_SORT;
          ofsort Bool
```

```
     (* Subtyping checked through subsetting of operations *)

            sub(adt1,adt2) = ops(adt2) Includes ops(adt1);

endtype (* DataType *)

(* Database is a repository for types. Operations are provided for
   adding types to and deleting types from the database; listing
   all types in the database; returning all subtypes of a given
   type in the database *)

type DB is DataType
     sorts DB
     opns empty    :              -> DB
          add_db   : DataType, DB -> DB
          del_db   : DataType, DB -> DB
          get_names: DB           -> Sort_List
          get_dt   : NameSort, DB -> DataType
          get_subs : NameSort, DB -> Sort_List
     eqns forall adt,adt1: DataType, sn: NameSort, adb: DB
          ofsort DB
            del_db(adt, adb)            = empty;
            (sub(adt,adt1) and sub(adt1,adt)) =>
            del_db(adt, add_db(adt1,adb)) = adb;
            (not(sub(adt,adt1) and sub(adt1,adt))) =>
            del_db(adt, add_db(adt1,adb)) = add_db(adt1,del_db(adt, adb));
            add_db(adt, add_db(adt, adb)) = add_db(adt, adb);
          ofsort Sort_List
            get_names(empty) = <>;
            get_names(add_db(adt,adb))    = (add_sort(IS_A(adt),<>)
                                              conc get_names(adb));
            get_subs(sn,empty)            = <>;
            (not(sub(adt,is_a(sn))))=>
            get_subs(sn,add_db(adt,adb))  = get_subs(sn,adb);
            (sub(adt,is_a(sn)))=>
            get_subs(sn,add_db(adt,adb))  = (add_sort(IS_A(adt),<>)
                                              conc get_subs(sn,adb));
          ofsort DataType
            (IS_A(adt) eq NULL_SORT)=>
            get_dt(sn,adb)                = NULL_DT;
            (IS_A(adt) eq sn)=>
            get_dt(sn,add_db(adt,adb))    = adt;
            (not(IS_A(adt) eq sn))=>
            get_dt(sn,add_db(adt,adb))    = get_dt(sn,adb);
endtype (* DB *)

(* Operations used by type manager *)

type Operation_Type is
     sorts operation_sort
     opns request, response,
          add_type, del_type,
          list_type, list_rel,
          all_rel: -> operation_sort
endtype (* operation_type *)


(* Identifiers of users of type manager *)

type id_type is
     sorts id_sort
     opns user_id1, user_id2: -> id_sort
endtype (* id_type *)

(* Error messages that type manager can return to the user *)

type error_type is NaturalNumber, Boolean
```

```
     sorts error_sort
     opns ok: -> error_sort
          requesting_entity_does_not_have_the_appropriate_authority:
                       -> error_sort
          properties_not_applicable_to_this_type_are_present:
                       -> error_sort
          type_name_not_registered:
                       -> error_sort
          type_name_already_registered:
                       -> error_sort
          relationship_not_allowed:
                       -> error_sort
          no_relationship_found_between_given_types:
                       -> error_sort
          no_compatible_type_found:
                       -> error_sort
          a_given_type_name_not_registered:
                       -> error_sort
          err_msg: Nat -> error_sort
     eqns ofsort error_sort
          err_msg(0) = ok;
          err_msg(succ(0)) =
            requesting_entity_does_not_have_the_appropriate_authority;
          err_msg(succ(succ(0))) =
            properties_not_applicable_to_this_type_are_present;
          err_msg(succ(succ(succ(0)))) =
            type_name_not_registered;
          err_msg(succ(succ(succ(succ(0))))) =
            type_name_already_registered;
          err_msg(succ(succ(succ(succ(succ(0)))))) =
            relationship_not_allowed;
          err_msg(succ(succ(succ(succ(succ(succ(0))))))) =
            no_relationship_found_between_given_types;
          err_msg(succ(succ(succ(succ(succ(succ(succ(0)))))))) =
            no_compatible_type_found;
          err_msg(succ(succ(succ(succ(succ(succ(succ(succ(0))))))))) =
            a_given_type_name_not_registered;
endtype (* error_type *)

behaviour
   hide a_check, db_type in

    test[add_t,del_t,list_t,add_r,del_r,list_r,all_r]
       |[add_t,del_t,list_t,add_r,del_r,list_r,all_r]|
    type_admin[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
       |[ a_check ]|
    authority_check[a_check]
where
     (* process for testing the functionality of the type manager *)

     process test[add_t,del_t,list_t,add_r,del_r,list_r,all_r]:noexit:=
        del_t !request !user_id1 !typename1;
        del_t !response !user_id1 ?msg: error_sort;
        add_t !request !user_id1 !typename1 !DT1;
        add_t !response !user_id1 ?msg: error_sort;
        add_t !request !user_id1 !typename2 !DT2;
        add_t !response !user_id1 ?msg: error_sort;
        add_t !request !user_id1 !typename3 !DT3;
        add_t !response !user_id1 ?msg: error_sort;
        del_t !request !user_id1 !typename1;
        del_t !response !user_id1 ?msg: error_sort;
        del_t !request !user_id1 !typename1;
        del_t !response !user_id1 ?msg: error_sort;
        add_t !request !user_id1 !typename4 !DT4;
        add_t !response !user_id1 ?msg: error_sort;
        list_r !request !user_id1 !typename1 !typename4;
        list_r !response !user_id1 ?msg: error_sort ?msg1: Bool;
```

```
        del_t !request !user_id1 !typename2;
        del_t !response !user_id1 ?msg: error_sort;
        list_r !request !user_id1 !typename1 !typename2;
        list_r !response !user_id1 ?msg: error_sort ?msg1: Bool;
        all_r !request !user_id1 !typename3;
        (
         all_r !response !user_id1 ?msg: error_sort; stop
        )
        []
        (
         all_r !response !user_id1 ?sl: Sort_List; stop
        )
     endproc (* test *)

     process type_admin[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
     :noexit:=

     let db_initial: DB = empty in

      type_repository[db_type](db_initial)
          |[ db_type ]|
      type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
where
      process type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
      :noexit:=

      op_type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
         []
      rel_type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]

where
    process op_type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
    :noexit:=

    add_type[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
            []
    del_type[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
            []
    list_type[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
where
       process add_type[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
       :noexit:=

         add_t !request ?id: id_sort             (* identifier of requester *)
                        ?stn: NameSort           (* type name *)
                        ?st: DataType;           (* type sort *)
       (
         a_check !request !add_type ?a_res: Nat;
        (
         (
          [a_res ne 0] ->
          add_t !response !id !err_msg(a_res);
          type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
         )
         []
         (
          [a_res eq 0] ->
          db_type !request !add_type !stn !st;
          db_type !response !add_type ?res: error_sort;
          add_t !response !id !res;
          type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
       )))
       endproc (* add_type *)

       process del_type[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
       :noexit:=
```

```
        del_t !request ?id: id_sort             (* identifier of requester *)
                       ?stn: NameSort;          (* type name *)
      (
        a_check !request !del_type ?a_res: Nat;
       (
        (
         [a_res ne 0] ->
         del_t !response !id !err_msg(a_res);
         type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
        )
        []
        (
         [a_res eq 0] ->
         db_type !request !del_type !stn;
         db_type !response !del_type ?res: error_sort;
         del_t !response !id !res;
         type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
      )))
      endproc (* del_type *)

      process list_type[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
      :noexit:=

        list_t !request ?id: id_sort             (* identifier of requester *)
                        ?stn: NameSort;          (* type name *)
      (
        a_check !request !list_type ?a_res: Nat;
       (
        (
         [a_res ne 0] ->
         list_t !response !id !err_msg(a_res);
         type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
        )
        []
        (
         [a_res eq 0] ->
         db_type !request !list_type !stn;
         db_type !response !list_type ?opl: OpSet ?res: error_sort;
      (* returns an empty operation list if non-applicable operation *)
         list_t !response !id !opl !res;
         type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
      )))
      endproc (* list_type *)
    endproc (* op_type_manager *)

    process rel_type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
    :noexit:=

      list_rel[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
           []
      all_rel[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
where
       process list_rel[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
       :noexit:=

         list_r !request ?id: id_sort           (* identifier of requester *)
                         ?sn1: NameSort         (* first type name *)
                         ?sn2: NameSort;        (* second type name *)
       (
         a_check !request !list_rel ?a_res: Nat;
        (
         (
          [a_res ne 0] ->
          list_r !response !id !err_msg(a_res);
          type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
         )
         []
```

```
        (
         [a_res eq 0] ->
         db_type !request !list_rel !sn1 !sn2;
         db_type !response !list_rel ?res1: error_sort ?res: Bool;
         (* checks two types are valid, if not, default of false for res *)
         list_r !response !id !res1 !res;
         type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
       )))
       endproc (* list_rel *)

       process all_rel[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
       :noexit:=

         all_r !request ?id: id_sort             (* identifier of requester *)
                        ?sn1: NameSort;          (* type name *)
       (
         a_check !request !all_rel ?a_res: Nat;
        (
         (
          [a_res ne 0] ->
          all_r !response !id !err_msg(a_res);
          type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
         )
         []
         (
          [a_res eq 0] ->
          db_type !request !all_rel !sn1;
          (
           (
            db_type !response !all_rel ?msg: error_sort;
            all_r !response !id !msg;
            type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
           )
           []
           (
            db_type !response !all_rel ?res: Sort_List;
            all_r !response !id !res;
            type_manager[add_t,del_t,list_t,add_r,del_r,list_r,all_r,db_type,a_check]
       )))))
       endproc (* all_rel *)
    endproc (* rel_type_manager *)
   endproc (* type_manager *)

    process type_repository[db_type](db: DB) :noexit:=

       add_type[db_type](db)
        []
       del_type[db_type](db)
        []
       list_type[db_type](db)
        []
       list_rel[db_type](db)
        []
       all_rel[db_type](db)
where
     process add_type[db_type](db: DB):noexit:=

      db_type !request !add_type ?stn: NameSort ?st: DataType;
     (
      [ not(is_in(stn, get_names(db))) ]->
      db_type !response !add_type !ok;
      type_repository[db_type](add_db(st,db))
      []
      [ is_in(stn, get_names(db)) ]->
      db_type !response !add_type !type_name_already_registered;
      type_repository[db_type](db)
     )
```

```
   endproc (* add_type *)

   process del_type[db_type](db: DB):noexit:=

    db_type !request !del_type ?stn: NameSort;
    (
     [ not(is_in(stn, get_names(db))) ]->
     db_type !response !del_type !type_name_not_registered;
     type_repository[db_type](db)
     []
     [ is_in(stn, get_names(db)) ]->
     db_type !response !del_type !ok;
     type_repository[db_type](del_db(get_dt(stn,db),db))
    )
   endproc (* del_type *)

   process list_type[db_type](db: DB):noexit:=

    db_type !request !list_type ?stn: NameSort;
    (
     [ not(is_in(stn, get_names(db))) ]->
     db_type !response !list_type !{} !type_name_not_registered;
     type_repository[db_type](db)
     []
     [ is_in(stn, get_names(db)) ]->
     db_type !response !list_type !ops(get_dt(stn,db)) !ok;
     type_repository[db_type](db)
    )
   endproc (* list_type *)

   process list_rel[db_type](db: DB):noexit:=

    db_type !request !list_rel ?sn1: NameSort ?sn2: NameSort;
    (
     [not(is_in(sn1,get_names(db))) or not(is_in(sn2,get_names(db))) ] ->
     db_type !response !list_rel !a_given_type_name_not_registered !false;
     type_repository[db_type](db)
     []
     [is_in(sn1,get_names(db)) and is_in(sn2,get_names(db)) ] ->
     db_type !response !list_rel !ok !sub(is_a(sn1),is_a(sn2))
                                (* or sub(is_a(sn2),is_a(sn1)) *);
     (* include above line if subtyping checked in both directions *)
     type_repository[db_type](db)
    )
   endproc(* list_rel *)

   process all_rel[db_type](db: DB):noexit:=

    db_type !request !all_rel ?sn: NameSort;
    (
     [ is_in(sn,get_names(db)) ] ->
     db_type !response !all_rel !get_subs(sn,db);
     type_repository[db_type](db)
     []
     [ not(is_in(sn,get_names(db))) ] ->
     db_type !response !all_rel !type_name_not_registered;
     type_repository[db_type](db)
    )
   endproc (* all_rel *)
  endproc (* type_repository *)
endproc (* type_admin *)

  process authority_check[a_check]
  :noexit:=
     a_check !request ?op: operation_sort !0;
     authority_check[a_check]
      []
```

```
       a_check !request ?op: operation_sort !succ(0);
       authority_check[a_check]
    endproc (* authority_check *)
endspec (* type_manager *)
```

Appendix B

Summary of Z Notation

This appendix provides a summary of the Z notation used in this thesis.

B.1 Schemas and Axiomatic Descriptions

Schemas have been used widely in this thesis. Their general format is:

Name

Declarations

Predicates

Axiomatic descriptions have been used widely. Their general format is:

Declarations

Predicates

B.2 Basic Types, Abbreviated Definitions and Free Types

Basic types have been used to represent information in Z that we have no wish to describe at a lower level of abstraction. Examples of these types include:

$$[TypeIdentifier; Label]$$

Abbreviated definitions have been used to introduce new types. Examples of these types include:

$$PList == \mathrel{_{seq}} Parameter$$

In this thesis usage has been made of basic free types and parameterised free types. Examples of these have included:

$$Causality ::= Client \mid Server$$

$$Action ::= Observable\langle ActSig \times State \times State \rangle \mid Internal\langle ActSig \times State \times State \rangle$$

B.3 Logic/Schema Calculus and Special Schema Operators

Logic/schema calculus and special schema operators have been used throughout this thesis.

true, false	logical constants	$\neg$ P	negation
P $\wedge$ Q	conjunction	P $\vee$ Q	disjunction
P $\Rightarrow$ Q	implication	P $\Leftrightarrow$ Q	equivalence
$\forall$ ...	universal quantification	$\exists$ ...	existential quantification
$\exists_1$ ...	unique existential quantification	Δ S	delta convention
Ξ S	xi convention	ΘS	Theta-expression
S[*a=b; c=d*]	Renaming	S $\setminus$ (a,b)	Hiding
S $\upharpoonright$ T	Projection	**pre** Op	Precondition
OpA $\fatsemi$ OpB	Sequential Composition	OpA $\gg$ OpB	Piping

B.4 Numbers and Arithmetic

The following lists the numeric sets and operations on them that have been used in this thesis.

$\mathbb{N}$	natural numbers	$\mathbb{Z}$	integers
+ - * div mod	arithmetic operations	= $\neq$ < $\leq$ $\geq$ >	arithmetic comparisons
$\mathbb{N}_1$	strictly positive integers	m . . . n	number range
#	size of a set	*min* S	minimum of a set
max S	maximum of a set		

B.5 Sets

The set operations and notation generally used in this thesis included:

$x \in S$	membership	$x \notin S$	non-membership
$\{ a,b, ..., z \}$	set display	$\{ x: T \mid P @ E \}$	set comprehension
(**let** x == E @ P)	Let expression	$\emptyset$ or $\{ \}$	empty set
$S \subseteq T$	subset relation	$S \subset T$	proper subset relation
$\mathbb{P} S$	power set	$\mathbb{P}_1 S$	non-empty power set
$\mathbb{F} S$	finite set	$\mathbb{F}_1 S$	non-empty finite set
$S \times T$	Cartesian product	(x,y,z)	tuple
first p	first of a pair	*second* p	second of a pair
$S \cup T$	set union	$S \cap T$	set intersection
$S \setminus T$	set difference	$\bigcup A$	generalised union
$\bigcap A$	generalised intersection		

B.6 Relations and Functions

The relations and functions used in the Z mathematical toolkit and employed to varying degrees in this thesis include:

$X \leftrightarrow Y$	binary relation	$x \mapsto y$	maplet
$\mathrm{dom}\ A$	domain of relation	$\mathrm{ran}\ A$	range of relation
$\mathrm{id}\ A$	identity relation	$Q \fatsemi R$	composition
$Q \circ R$	backwards composition	$S \lhd R$	domain restriction
$S \rhd R$	range restriction	$S \ndres R$	domain anti-restriction
$S \nrres R$	range anti-restriction	$R^{\sim}$	relational inverse
$R(S)$	relational image	$Q \oplus R$	overriding
$R\,k$	iteration	R^+	transitive closure
R^*	reflexive transitive closure	f(x)	function application
$X \rightarrow Y$	partial functions	$X \rightarrow Y$	total function
$X \rightarrowtail Y$	partial injection	$X \rightarrowtail Y$	total injection
$X \twoheadrightarrow Y$	partial surjection	$X \twoheadrightarrow Y$	total surjection
$X \rightarrowtail\!\!\!\!\twoheadrightarrow Y$	bijection	$X \pfun Y$	finite partial function
$X \finj Y$	finite partial injection		

B.7 Sequences

The sequence operations that can be found in the Z mathematical toolkit and have been used in various parts of this thesis include:

seq	finite sequences	seq_1	non-empty sequences
iseq	injective sequences	$\langle a; ::z \rangle$	sequence display
$s \frown t$	concatenation	$rev\ s$	reverse
$head\ s$	head of a sequence	$last\ s$	last element of a sequence
$tail\ s$	tail of a sequence	$front\ s$	all but last element of sequence
$s \upharpoonright U$	filtering	$s \mathrm{prefix} t$	prefix relation
$s \mathrm{suffix} t$	suffix relation	$s \mathrm{in} t$	segment relation
$\mathrm{disjoint}\ SS$	disjointness	$SS \mathrm{partition} T$	partition relation
$U \upharpoonleft s$	extraction		

Appendix C

Summary of LOTOS Notation

This appendix provides a brief summary of the LOTOS notation used in this thesis.

C.1 An Overview of LOTOS

A LOTOS specification typically consists of two main parts: a data typing part based on Act One [65] and a behavioural description part given in the process algebra. The process algebra itself is related (but not the same as) Milner's Calculus of Communicating Systems [141] and Hoare's Communicating Sequential Processes [96]. The similarities and differences between LOTOS and these other two languages are given in [71].

C.2 An Overview of the Act One Part

LOTOS presents data as abstract data types (ADTs) using Act One. The description of an ADT typically consists of three parts:

- the new type name and types used in its construction;
- the new sorts introduced into the specification and the operations that are applicable to these sorts;
- the equations associated with these sorts.

One example of an abstract data type is:

```
type (* new type name *) is (* imported types *)
     sorts (* sortname(s) *)
     opns  (* operations on sortname(s) *)
     eqns  (* equations associated with operations *)
endtype
```

Here the new type name represents the type of the ADT being defined. This should be unique to the specification. The reserved word *is* may be followed by zero or more type definitions. The sorts gives the name of the data sorts that are being introduced to the specification. The operations on the sort provide the signature for the operations available on that sort. Operations are defined as functions with zero or more inputs and a single output. Operations with zero inputs are often referred to as constants.

The equations associated with an ADT show what the effect is of operations occurring. Typically, this effect is given by providing some form of equivalence class of equational terms, e.g. a + 0 = a. The equations themselves are grouped according to the result sorts that they produce.

An example of an ADT defined in Act One is a stack. This can be represented as:

```
type Stack is NaturalNumber
     sorts Stack
     opns  create : -> Stack
           push   : Nat, Stack -> Stack
           pop    : Stack -> Nat
     eqns  forall s: Stack, n: Nat
           ofsort Nat
             pop(push(n,s)) = n;
             pop(create)    = 0;
endtype (* Stack *)
```

The constant "0" here is used to indicate that an error has occurred.

As well as direct inclusion following the *is* keyword, Act One allows re-use of data definitions through parameterisation, renaming and actualisation. An example of a data type that can be parameterised might be:

```
type AbstractStack is
     formalsorts data
     formalopns d: -> data
     sorts Stack
     opns  create : -> Stack
           push   : data, Stack -> Stack
           pop    : Stack -> data
     eqns  forall s: Stack, d: data
           ofsort data
             pop(push(d,s)) = d;
             pop(create)    = 0;
endtype (* AbstractStack *)
```

This can be actualised by providing sorts and operations for the formal sorts and operations given. One example of this actualisation might be:

```
type RealStack is AbstractStack
     actualizedby (* some existing data type *)
     using sortnames (* some sort associated with existing data type *) for data
           opnnames (* some operation on existing sort *) for d
endtype (* RealStack *)
```

We note here that the actual operation must satisfy all constraints associated with the formal operation, *e.g.* the actual and formal operations should be structurally identical. Further, if formal equations are given then these must be satisfied by the actual operation supplied.

Renaming may be used when a new data definition is required that is similar to an existing data type. Renaming can be achieved by explicitly changing the sort names and operation names of the existing data type. The syntax for renaming is given by:

```
type (* new type *) is (* existing type *) renamedby
     sortnames (* new sort name *) for (* old sort name *)
               ... (* other sorts renamed *)
     opnnames (* new operation name *) for (* old operation name *)
               ... (* other operations renamed *)
endtype (* new type *)
```

We note here that actualisations and renamings differ in that renamed data types must be isomorphic, *i.e.* they may not extend the signature of the existing data type. Actualisations may do so.

C.3 An Overview of the Process Part

The process algebra part of a LOTOS specification consists of a collection of process abstractions (definitions). These may be regarded as black boxes composed of observable and internal actions. Observable actions (interactions) take place through synchronisation with the environment of the process. Internal actions do not require synchronisation with the environment. The processes themselves are defined as

Name	Syntax
inaction	**stop**
termination	**exit**
termination with parameters	**exit(**$e_1 \ldots e_n$**)**
choice	B_1 [] B_2
action prefix	g; B
unobservable action prefix	**i**;B
full parallel composition	$B_1 \parallel B_2$
general parallel composition	$B_1 \mid [g_1; \ldots g_n] \mid B_2$
interleaving	$B_1 \mid\mid\mid B_2$
hiding	**hide** $g_1; \ldots g_n$ **in** B
process instantiation	P[$g_1; \ldots g_n$]($e_1; \ldots e_n$)
guarding	[Ce]$\rightarrow$ B
disabling	B_1[> B_2
enabling	$B_1 \gg B_2$
enabling with value passing	$B_1 \gg$ **accept** $v_1 : T_1$ **in** B_2
$B; B_i$: behaviour expressions	e_1: value expressions
$g; g_i$: gate identifiers	P : process identifier
Ce : conditional expression	v_i: variable identifiers

Table C.1: Syntax of LOTOS Behaviour Operators

temporally ordered collections of actions. These temporal orderings are given in a *behaviour expression*. The actual temporal orderings that exist is dependent upon the behavioural composition operators used. A summary of the behavioural composition operators is given in table C.1.

Action templates are modelled in LOTOS by event offers. Internal events are modelled either through the internal event symbol **i** or through event offers at hidden gates. Event offers may be represented by gates or by gates followed by one or more associated arguments. LOTOS provides two ways to associate arguments with an event offer. Arguments can be preceded by **!a** in which case the argument is a single value expression. Alternatively, arguments can be preceded by **?a:A** in which case **a** is a variable of the sort **A**. For example, the event offer **g !0 ?x: Bool** indicates that the natural number value 0 is required and the term **x: Bool** indicates that any value of sort Boolean, *i.e.* true or false, is acceptable and assigned to **x**.

Events may occur provided all event offers involved in the synchronisation have a non-empty intersection of associated argument values. It is possible also to restrict the values that can be accepted by event offers through selection predicates, *e.g.* the event offer **g ?x:Nat[x>2];** indicates that the event offer will only synchronise with other event offers that offer natural number argument values greater than 2.

Processes have the following structure:

```
process PROCESS_NAME[ gate-list ](parameter-list): functionality:=
     (* behaviour expression *)
     where
         (* data type definitions *)
         (* process definitions *)
endproc
```

The functionality can be either **noexit**, in which case the process may not successfully terminate, or **exit** in which case the process may successfully terminate. Process with **noexit** functionality are frequently recursively defined processes or processes that offer the null behaviour **stop**, *i.e.* do nothing. Processes that may terminate can do so with results. The types (sorts) of these results have to be declared when the process is defined.

C.4 Specification Structure

A LOTOS specification typically has the following structure:

```
specification SPECIFICATION_NAME[ gate-list ](parameter-list):functionality
library ... endlib
type ... endtype
type ... endtype
...
behaviour ...(* behaviour expressions *)
where
     process ... endproc
     process ... endproc
     ...
endspec
```

In the library the more commonly used data types are given, *e.g.* the data types found in the LOTOS standard [101].

There exists a rich source of tutorial material on LOTOS that greatly expands on this brief overview. Examples of these tutorials include: [23, 131, 59, 197].